xBS-GNN: Accelerating Billion-Scale GNN Training on FPGA

Yi-Chien Lin*
University of Southern California
Los Angeles, California
yichienl@usc.edu

Zhijie Xu*
University of Michigan
Ann Arbor, Michigan
zhijiexu@umich.edu

Viktor Prasanna
University of Southern California
Los Angeles, California
prasanna@usc.edu

***Abstract*—Graph Neural Networks (GNNs) have been successfully used in a variety of challenging application areas, including Electronic Design Automation and molecular property prediction, among others. However, training GNN models is time-consuming as it incurs a high volume of irregular data accessing due to its graph-structured input data; such a challenge is further exacerbated in real-world applications as they often involve training GNN models on large-scale graphs with over billions of edges. While several GNN accelerators have been proposed, most of them cannot scale to billion-scale graphs due to the limitation of memory capacity. To this end, we propose xBS-GNN, a novel accelerator optimized for billion-scale GNN training. xBS-GNN exploits the multi-level memory hierarchy on state-of-the-art FPGA-based systems to enable billion-scale GNN training. To achieve high training throughput, xBS-GNN jointly exploits several optimizations, including (1) a novel data placement policy optimized for GNN training, along with (2) a vertex-renaming technique and memory-efficient lookup table design for fast data retrieval, and (3) a feature quantization mechanism to reduce memory traffic. We evaluate xBS-GNN with a three-layer GCN model on three large datasets. xBS-GNN achieves up to 8.39× speedup over a widely-used GPU baseline and up to 5.13× speedup over a state-of-the-art GNN training accelerator. xBS-GNN also demonstrates high scalability on multi-FPGA platforms.**

***Index Terms*—GNN training, Accelerator, FPGA**

I. Introduction

Graph Neural Network (GNN) is a widely used Machine Learning model that extracts useful information from graph-structured data. GNNs have been applied in many challenging areas, such as Electronic Design Automation (EDA) [1], social recommendation systems [2], [3], molecular property prediction [4], [5], among others. Despite its usefulness, training a GNN model for these real-world applications is time-consuming as it often involves large-scale graphs with billions of edges, incurring massive amount of computations and high volume of irregular data accessing. While several GPU-based solutions have been proposed [6]–[8], GPU platforms suffer from low resource utilization, resulting in limited acceleration; this is because the fixed datapath and cache design of GPU cannot efficiently process graph-structured data, leading to a low cache hit rate [9]. Another line of work [10]–[13] proposed to accelerate GNN training using FPGA platforms, and leverage customized hardware designs to address the inefficiency of irregular data access. However, to achieve high performance, FPGA accelerators typically assume the input graph can be entirely stored in the FPGA off-chip memory (e.g., FPGA DDR or HBM). Consequently, these accelerators cannot train on billion-scale graphs used in real-world applications as they exceed the size of the available FPGA off-chip memory. For example, FPGAs provide 16-64 GBs of DDR memory, whereas billion-scale graphs like MAG240M [14] surpass 200 GBs in size. Storing such large graphs in the CPU main memory can overcome this limitation, but would lead to poor training performance as a high volume of data needs to be transferred through PCIe, which has limited memory bandwidth.

Motivated by these challenges, we propose xBS-GNN, a novel hardware accelerator optimized for billion-scale GNN training. Recognizing that billion-scale graphs surpass the capacity of FPGA off-chip memory, we introduce a novel data placement policy that selectively stores portions of the graph in the FPGA on-chip SRAM and its off-chip memory, while keeping the remainder in the CPU main memory. We depict the multi-level memory system of xBS-GNN in Figure 2. Furthermore, since traditional caching policy fails to capture the characteristic of GNN's irregular data access [9], we develop a Pre-sampler to estimate the access frequency of each node, guiding our data placement policy to prioritize the storage of frequently accessed nodes. The data placement policy allows xBS-GNN to scale to billion-scale graphs while still achieving high performance by maximizing on-chip data access and minimizing the expensive CPU-FPGA data transfer via PCIe. xBS-GNN leverages the fine-grained hardware control offered by FPGAs to deploy a novel memory organization and data placement policy optimized for GNN training; such optimizations are not feasible on general-purpose processors and GPUs due to their fixed hardware design. Since the data is stored across multiple levels of memory, we utilize a vertex-renaming technique along with a memory-efficient lookup table design for fast data retrieval. In addition, xBS-GNN exploits feature quantization that quantizes the node features from FP32 to INT8. Feature quantization offers several advantages: (1) By compressing the features, it enables xBS-GNN to allocate more of the frequently accessed nodes in the on-chip SRAM or FPGA off-chip memory, and (2) it reduces the PCIe traffic for accessing data stored in the CPU main

*Both authors contributed equally to this research.

TABLE I
NOTATIONS OF GNN

Notation	Description	Notation	Description
$\mathcal{G}(\mathcal{V},\mathcal{E})$	graph topology	$\boldsymbol{h}_i^l$	feature vector of v_i at layer l
$\mathcal{V}$	set of nodes	$\boldsymbol{a}_i^l$	aggregation of v_i at layer l
$\mathcal{E}$	set of edges	L	number of GNN layers
$\boldsymbol{X}$	input feature matrix	$\boldsymbol{W}^l$	weight matrix of layer l
$\mathcal{V}^l$	sampled nodes at layer l	$\mathcal{N}(i)$	neighbors of v_i
$\mathcal{E}^l$	sampled edges at layer l	$\phi(.)$	element-wise activation

memory; (3) It also increases hardware parallelism as modules implemented in INT8 are more resource-efficient. Finally, we evaluate the scalability of xBS-GNN on multiple FPGAs. xBS-GNN demonstrates high scalability on billion-scale graphs. The key contributions of this work are:

- We propose xBS-GNN, a novel GNN training accelerator optimized for billion-scale GNN training; xBS-GNN offers high training throughput by jointly exploring several effective optimizations.
- We propose a data placement policy, tailored for GNN training, to store billion-scale graphs in the multi-level memory of FPGAs, and a memory-efficient lookup table for fast data retrieval from multiple levels of the memory.
- We jointly exploit feature quantization with our data placement policy, further boosting the training performance of xBS-GNN by up to 4.56× while retaining similar accuracy as using FP32.
- We evaluate xBS-GNN on three large datasets. xBS-GNN achieves up to 8.39× speedup compared with a widely-used GPU baseline, and up to 5.13× over the state-of-the-art multi-GPU training accelerator.
- We show that xBS-GNN offers high scalability for billion-scale GNN training on multi-FPGA platforms.

II. BACKGROUND

A. Graph Neural Networks

We define the notations related to a GNN in Table I. Graph Neural Networks (GNNs) are a class of neural networks specifically designed to process data structured in graph form, utilizing node-related features as their input. By aggregating information across the graph's structure (i.e., feature aggregation) and subsequently transforming these features into a latent space (i.e., feature transformation), GNNs generate node representations that contain higher-order neighbor information. In general, a GNN model can be expressed using the aggregate-transform paradigm [15]:

$$\boldsymbol{a}_v^l = \text{AGGREGATE}(\boldsymbol{h}_u^{l-1} : u \in \mathcal{N}(v) \cup \{v\}) \quad (1)$$

$$\boldsymbol{h}_v^l = \phi(\text{TRANSFORM}(\boldsymbol{a}_v^l, \boldsymbol{W}^l)) \quad (2)$$

During the feature aggregation stage, for each node v, the feature vectors h_u^{l-1} of the neighbor nodes $u \in \mathcal{N}(v)$ are aggregated into a_v^l using model-specific operators such as mean, max, or LSTM. Since graph-structured data are non-Euclidean, accessing the feature vectors h_u^{l-1} of the neighbor nodes incurs a high volume of irregular data access. The feature transformation stage performs a multi-layer perceptron (MLP) followed by an activation function ϕ (e.g., ReLU).

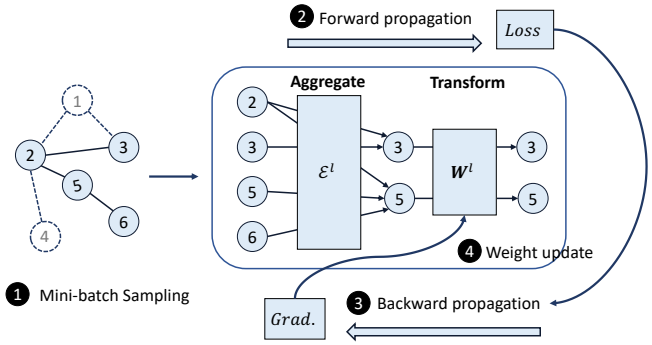

Fig. 1. Workflow of sampling-based GNN training

GNNs are originally trained with the full graph [16], which takes the entire graph $\mathcal{G}(\mathcal{V},\mathcal{E})$ and the input feature matrix $\boldsymbol{X}$ as input. However, this approach suffers from low scalability as it incurs substantial memory overhead as the graph size increases, and sampling-based GNN training [5] is proposed to overcome this challenge. Instead of training on the entire graph, sampling-based GNN training only trains on a mini-batch (i.e., sampled sub-graph) in each iteration, leading to less memory overhead. The methodology of selecting the nodes and edges to produce a mini-batch depends on the sampling algorithm [5], [17]–[19]. For example, the Neighbor Sampling algorithm [5] randomly selects n^l neighbors for each node for each layer, where n^l is a predefined budget on the sampling size for layer l. We depict the general workflow of sampling-based GNN training in Figure 1: First, a mini-batch is sampled and used as the input of GNN model propagation; GNN model propagation includes feature aggregation and feature transformation. After the backpropagation is completed, the gradient is derived, which is then used to update the model.

B. Related Work

1) GPU-based GNN Accelerator: Several works have been proposed to accelerate GNN training on CPU or GPU platforms [6], [7], [20]–[22]. Some of these works, like DistDGLv2 [6] and P^3 [7], train the GNN model on a distributed GPU platform, allowing them to support training on billion-scale graphs by utilizing the large memory resources. They adopt graph partitioning algorithms like METIS [23] to partition the graph and store one partition in each machine. These works focus on (1) balancing the workload and (2) minimizing the data communication overhead among the machines. PaGraph [8] trains the GNN model on a multi-GPU platform and proposes to store nodes with high out-degree in the GPU global memory to reduce CPU-GPU data transfer overhead. However, as mentioned in Section I, due to the fixed datapath and cache design, GPUs cannot efficiently access

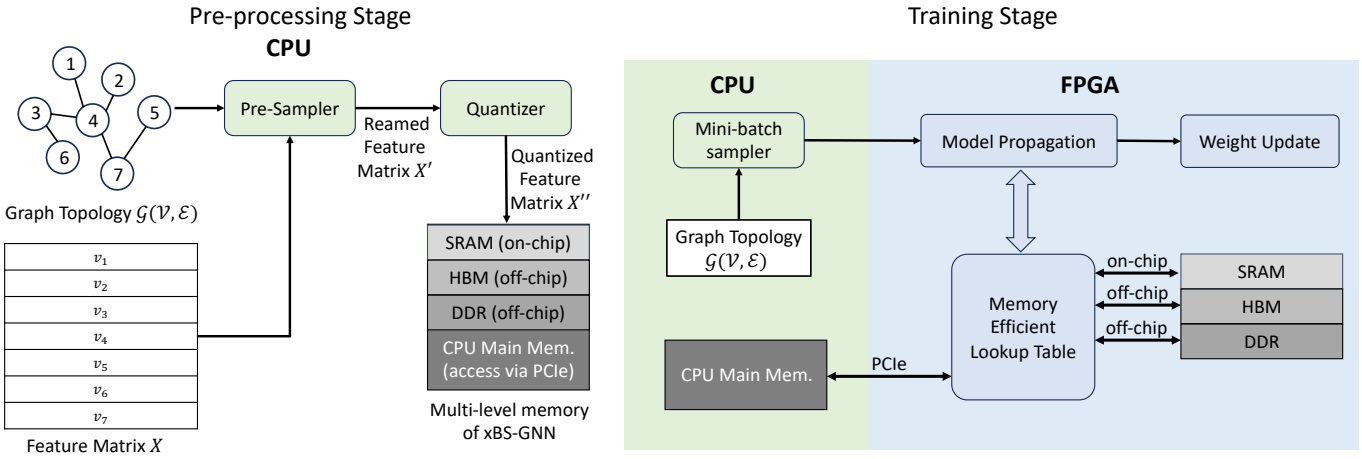

Fig. 2. System overview of xBS-GNN

graph-structured data. Thus, GPU-based solutions suffer from low resource utilization, leading to limited acceleration. To overcome this challenge, another line of research seeks for hardware-based solutions.

2) GNN Hardware Accelerator: Several hardware accelerators have been proposed to accelerate GNN computations [11]–[13], [24]–[26]. They develop dedicated hardware to accelerate GNN operations or to reduce irregular memory access overhead of the graph-structured data. However, most of the accelerators [24]–[26] only support GNN inference acceleration. Few works like GraphACT [12], HP-GNN [13], and Rubik [27] supports GNN training; nevertheless, works like GraphACT and HP-GNN assume the graph is entirely stored in the FPGA DDR memory, rather than the CPU main memory. Consequently, these accelerators cannot train on billion-scale graphs, which exceed the size of the FPGA DDR. Rubik supports loading data from off-chip memory, which theoretically enables it to support billion-scale GNN training. However, the largest dataset used for evaluation in Rubik, the Reddit [17] dataset, is only within the million-scale range, and the achievable performance of Rubik when scaling to billion-scale graphs is uncertain. Supporting billion-scale graphs is important as they are widely used in real-world applications [14].

3) Quantization: Quantization is a technique commonly used in both ML training and inference. While several works [28]–[31] have been proposed, they primarily focus on quantizing the model weights and activations in Deep Neural Networks as they account for most of the memory overhead; however, these works are less applicable to GNNs because GNN model weights are relatively small, often just a few megabytes [32]. Instead, the memory overhead is mainly caused by the feature vectors of the input graph [32]. Therefore, GNNs require a distinct quantization approach tailored to their unique characteristics. Several works [32]–[35] have developed quantization techniques tailored for GNN to reduce memory overhead or data traffic across distributed nodes. Nevertheless, most of these works primarily focus on exploring quantization alone, such as how to adaptively quantize the data during training to minimize memory overhead while preserving high model accuracy, and do not jointly explore other optimizations for further acceleration. Note that the focus of xBS-GNN is not to introduce a new quantization technique, but to propose a novel accelerator design that can support billion-scale GNN training and to demonstrate the performance improvements that can be achieved by jointly exploiting several effective optimizations, including quantization.

III. System Design

A. Overview

We depict the overview of our system design in Figure 2. xBS-GNN operates in two stages: the pre-processing stage and the training stage. The pre-processing stage, which runs on the CPU platform, is only performed once. The training stage, on the other hand, is performed multiple times on both CPU and FPGA platforms until the training is concluded (e.g., when model converges). During the pre-processing stage, xBS-GNN performs a step called *pre-sampling* to identify the access frequency of each node, and rename the nodes based on their access frequency (Section III-B). Additionally, each feature vector is quantized from FP32 to INT8 (Section III-D) before being stored into the multi-level memory. Note that for billion-scale graphs, the overhead of loading the node features bottlenecks GNN training [33], and requires careful optimizations. Therefore, the focus of the pre-processing stage is to efficiently store the node features in the multi-level memory to minimize the data transfer overhead. During the training stage, xBS-GNN performs sampling-based GNN training (details in Figure 1.) xBS-GNN assigns the CPU platform to perform mini-batch sampling, since CPU platform can flexibly support a variety of sampling algorithms. On the other hand, xBS-GNN assigns the FPGA platform to perform the model propagation and model weight update, because FPGA can efficiently address the irregularity in GNN operations using dedicated hardware kernels (Section III-E). During model propagation, the hardware kernels fetch node features from the multi-level

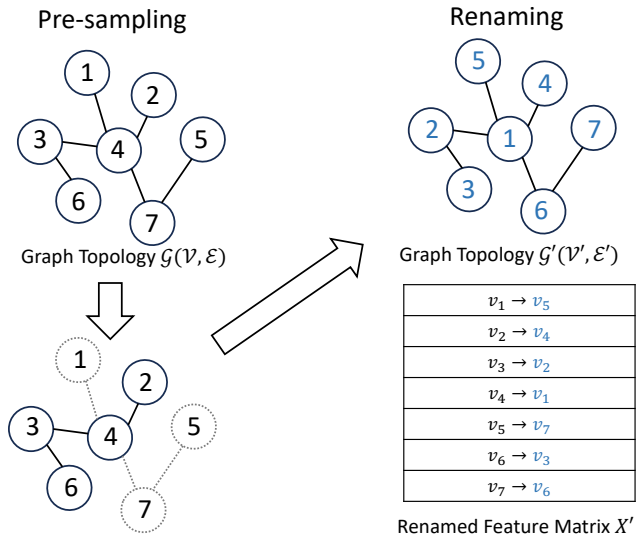

Fig. 3. Pre-sampler sorts and renames the nodes based on access frequency

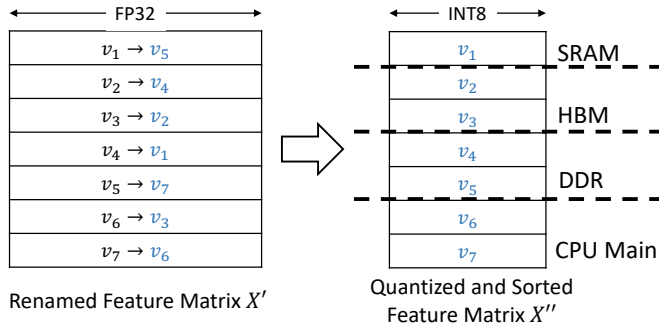

Fig. 4. Storing the pre-processed feature matrix in the multi-level memory

memory to perform feature aggregation; in order to identify where the required node feature resides (i.e., in which level of memory hierarchy), we developed a memory-efficient lookup table (Section III-C) for fast data retrieval.

B. Pre-sampler

Determining the data placement policy for GNN training is non-trivial due to the intricate nature of graph accessing; [9] shows that traditional caching policy cannot capture the graph access pattern, leading to a low cache hit rate. Thus, we develop a Pre-sampler to estimate the access frequency of each node; this allows us to place the most frequently accessed nodes into the on-chip SRAM, followed by HBM, DDR, and finally, the CPU main memory. The Pre-sampler works in two steps: pre-sampling and node renaming. Pre-sampling runs the sampling algorithm on a given dataset for several iterations prior to the actual training. Note that pre-sampling is a lightweight operation as it only operates on the graph topology, and does not involve loading the node features. While most sampling algorithms perform a random shuffling at the beginning of the sampling, we observe that the access frequency of each node remains similar in each iteration. Therefore, we can approximate the access frequency of each node by performing pre-sampling for several iterations and taking the average of the results. Pre-sampling generates multiple mini-batches that are used to calculate the access frequency of each node v_i; the access frequency is defined as $(\#$ of access of node $v_i)/(\Sigma_{v_j \in \mathcal{V}}$ $\#$ of access of node $v_j)$. The Pre-sampler sorts the nodes based on access frequency and performs node renaming using the ranking of their access frequency. For example, in Figure 3, assume node 4 has the highest access frequency, followed by node 3. Then, they will be renamed as node 1 and node 2, respectively. Node renaming allows xBS-GNN to place node features into the multi-level memory based on access frequency, by starting with nodes that have the smallest labeled numbers.

C. Memory-efficient Lookup Table

While storing the feature matrix in the multi-level memory with our data placement policy allows xBS-GNN to access data more efficiently, it also comes with the challenge that the feature vectors are now distributed across multiple levels of memory. A standard way to retrieve data from a multi-level memory is to send a query to each memory level to locate the data. However, this process results in a noticeable amount of access overhead for GNN training [13]. Another approach is to use key-value pairs for data retrieval (e.g., KVStore used in DistDGL [36]); but this leads to additional memory overhead to store the key-value pairs, which can be prohibitively expensive for billion-scale graphs. We propose a memory-efficient lookup table design for data retrieval. As mentioned in Section III-B, the feature matrix is renamed based on the ranking of the accessed frequency, and then sorted based on the renamed node numbers. Shown in Figure 4, the node features are stored into the multi-level memory, starting with the nodes with the smallest labeled number. Therefore, the lookup table can simply record the dividing line of each level for data retrieval. For example, in Figure 4, v_1 and above are stored in the SRAM, v_3 and above (before v_1) are stored in the HBM, and so on and so forth. xBS-GNN can locate the node feature by comparing the query node number with the dividing lines (e.g., v_1, v_3, v_5 in Figure 4). Thus, the memory-efficient lookup table only records the dividing lines, and each retrieval only requires a comparison between the query node number and the dividing lines. Note: FPGAs have multiple banks of DDR and HBM. xBS-GNN stores the node features in a cyclic order across these banks to balance the data traffic; if the node features are stored sequentially, the first bank will be populated with the most frequently accessed node, leading to imbalanced traffic among the banks.

D. Feature Quantizater

xBS-GNN features a Feature Quantizer which quantizes the feature matrix $\boldsymbol{X}$ from FP32 to INT8 format using affine quantization. Quantization is often used to reduce memory requirements and improve performance [37], [38]; more importantly, it can be synergistically used with xBS-GNN's data

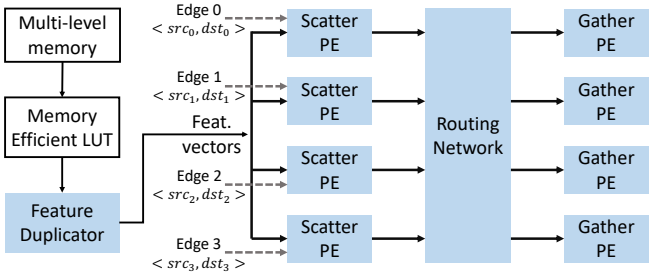

Fig. 5. Design of the feature aggregation kernel (blue boxes)

placement policy, bringing higher performance improvement than applying quantization alone. Specifically, compressing the data into INT8 format allows xBS-GNN to place more frequently accessed nodes in the on-chip SRAM and the FPGA off-chip memory, compared with data stored in FP32. Also, it increases hardware parallelism as kernels implemented in INT8 are more resource-efficient than in FP32. [39] shows that a single DSP can perform two parallel multiply–accumulate (MAC) operations in INT8, but can only perform one MAC operation in FP32.

While quantization often degrades accuracy, it has been shown that GNNs have a higher tolerance to quantization error than traditional Deep Neural Networks because during feature aggregation (Section II), the node features are aggregated, and the quantization error can be canceled out during the process [33]. Thus, quantizing the node feature does not lead to a noticeable impact on the accuracy.

E. Hardware Kernels for GNN Model Propagation

xBS-GNN features dedicated hardware kernels optimized for GNN operations (i.e., aggregation and transformation). Drawing inspiration from HP-GNN [13] and ThunderGP [40], the feature aggregation kernel in xBS-GNN adopts a similar design philosophy to maximize data reuse. As shown in Figure 5, the kernel adopts a scatter-gather model design, which is widely used in the field of graph processing [40], [41]. xBS-GNN stores the graph topology using the COOrdinate (COO) format, where each edge is stored as a pair of source and destination nodes. To maximize data reuse, the edges are sorted by the source nodes. This ensures that edges sharing the same source are processed concurrently, preventing the kernel from repeatedly loading the same node feature. Specifically, a feature vector of a source node is loaded from the multi-level memory and then broadcast to each Processing Element (PE) through the Feature Duplicator. In each iteration, a distinct edge is loaded into each PE. If the source node of the loaded edge aligns with the currently loaded feature, the PE reuses it for computation. If not, the PE awaits the Feature Duplicator's broadcast of the next source node's feature vector to replace the existing one. Since the edges are sorted, all edges with the same source nodes are guaranteed to be executed before the feature in the PE is replaced with the next source node. Maximizing data reuse (i.e., minimizing memory traffic) is advantageous when the node is located in the FPGA DDR or the CPU main memory, as accessing these memory levels incurs significant overhead. For the feature transformation kernel, we adopt a systolic-array-based design to perform the multi-layer perceptron with a 2-D MAC array. The feature aggregation kernel is implemented in INT8 to aggregate the quantized feature vectors. After the aggregation, the data is dequantized to FP32 for feature transformation as xBS-GNN keeps the model weights in FP32.

TABLE II
SPECIFICATIONS OF THE PLATFORMS

Platforms	CPU	FPGA	GPU
Devices	Intel Xeon 6326	AMD Alveo U280	Nvidia RTX A5000
Technology	10 nm	16 nm	8 nm
Frequency	2.9 GHz	300 MHz	2000 MHz
Peak Performance	537 GFLOPS	24.5 TFLOPS	27.8 TFLOPS
On-chip Memory	24 MB L3	41 MB	6 MB L2
Memory Bandwidth	171 GB/s	460 GB/s	768 GB/s

TABLE III
STATISTICS OF THE DATASETS

Dataset	**#Vertices**	**#Edges**	f_0	f_L
ogbn-products	2,449,029	61,859,140	100	47
ogbn-papers100M	111,059,956	1,615,685,872	128	172
MAG240M	244,160,499	1,729,762,391	768	153

IV. EXPERIMENTS

A. Experimental Setup

1) Environment: We evaluate xBS-GNN on an Alveo U280 FPGA card connected to a dual-socket Xeon 6326 CPU via PCIe. We implement xBS-GNN using Vitis 2023.1 with High-Level Synthesis (HLS). Our implementation uses 71% of the LUTs, 72% of the DSPs, 86% of the BRAMs, and 82% of the URAMs on the U280. The FPGA kernels run at 300 MHz. We compare xBS-GNN with a GPU baseline that runs on a Nvidia A5000 GPU connected to a dual-socket Xeon 6326 CPU. The GPU implementation uses Python 3.9, PyTorch 2.0 and Deep Graph Library 2.0. We list the platform details in Table II.

2) GNN model and Dataset: To generate mini-batches for sampling-based GNN training, we adopt the Neighbor Sampler [5], and train on a widely-used three-layer GCN [16] model, which has the sampling size of [15, 10, 5] for each layer, and the hidden feature size is 128. For datasets, we choose two billion-scale datasets: ogbn-papers100M [42], and MAG240M [14]. We also choose the ogbn-products dataset with tens of millions of edges for comparison with existing works. We list the details of the datasets, and the dimensions of the input layer and output layer in Table III.

B. Accuracy

To evaluate the accuracy difference of training with node features stored in FP32 and in INT8, we run the GNN training for 20 epochs and measure the model accuracy under the

TABLE IV
ACCURACY (%) COMPARISON OF FEATURES IN FP32 AND INT8

Data Type	ogbn-products	ogbn-papers100M	MAG240M
FP32	93.0%	67.4%	59.1%
INT8	93.7% (+0.7%)	67.7% (+0.3%)	58.1% (-1.0%)

TABLE V
COMPARISON WITH STATE-OF-THE-ART (EPOCH TIME (SEC.))

	ogbn-products	ogbn-papers100M	Geo. Mean Speedup
PaGraph	1.18 (1.00×)	4.00 (1.00×)	1.00×
xBS-GNN	0.23 (5.13×)	3.21 (1.25×)	2.53×
DistDGL_v2	0.3 (1.00×)	4.16 (1.00×)	1.00×
xBS-GNN	0.62 (0.48×)	2.53 (1.64×)	0.89×

two data types. Since all three datasets chosen are multi-class classification problem, the accuracy is defined as the proportion of correctly classified cases. As shown in Table IV, training with INT8 results in a maximum accuracy loss of 1%; in some cases, there is even a slight accuracy gain, which shows that training with features in INT8 leads to comparable accuracy as in FP32.

C. Performance Evaluation

1) Comparison with State-of-the-art: We compare xBS-GNN with two state-of-the-art GNN accelerators that support large-scale GNN training: DistDGL_v2 [6] and PaGraph [8]. We measure the execution time of the training stage (Figure 2) for comparison, as the pre-processing overhead is a one-time cost that can be amortized. For example, on MAG240 dataset, the pre-processing overhead is less than a minute, which accounts for less than 3% of the total training time (assuming training until the model converges, which is around 20 epochs.) PaGraph adopts a two-layer GCN model with sampling size [25, 10], and DistDGL_v2 [6] adopts a three-layer GCN model with sampling size [15, 10, 5]. To perform a fair comparison, we adjust our setup accordingly. Note that both PaGraph and DistDGL_v2 are executed on platforms with multiple GPUs: PaGraph runs on a platform equipped with **8 Nvidia V100 GPUs**, and DistDGL_v2 runs on a platform with **64 Nvidia T4 GPUs**. Consequently, these accelerators have access to significantly more computational and memory resources than xBS-GNN, which runs on a platform with a single FPGA. Compared with PaGraph, xBS-GNN achieves 2.53× speedup w.r.t. the geometric mean. While PaGraph also places a portion of node features in the GPU global memory, the placement policy is simply based on node degree, leading to a lower hit rate than xBS-GNN. Compared with DistDGL_v2, xBS-GNN can achieve 89% of its performance using much less resources. DistDGL_v2 trains on a distributed platform, which incurs inter-machine communication overhead. As mentioned in Section II, state-of-the-art hardware accelerators [12], [13] do not support training on billion-scale graphs; thus, we only compare the performance of xBS-GNN with HP-GNN [13] using two million-scale graphs: Flickr and Yelp [17], on a two-layer GCN model. Compared with HP-GNN, xBS-GNN achieves 1.5× and 2.1× speedup on Flickr and Yelp datasets, respectively. This shows that while xBS-GNN is optimized for training on billion-scale graphs, it also offers high performance for training on million-scale graphs.

TABLE VI
CROSS PLATFORM COMPARISON (EPOCH TIME (SEC.))

	ogbn-products	ogbn-papers100M	MAG240M
DGL	4.81 (1.00×)	21.35 (1.00×)	136.98 (1.00×)
xBS-GNN	0.62 (7.77×)	2.53 (8.39×)	56.61 (2.42×)

2) Cross Platform Comparison: We compare the GNN training performance of xBS-GNN with Deep Graph Library (DGL) [43], a state-of-the-art GNN library for CPU/GPU platforms. Different from Table V, where we compare xBS-GNN with works that run on platforms with multiple GPUs, we run the DGL baseline on a single RTX A5000 GPU. Note that for the GPU baseline, we also quantize the node features into INT8 format for a fair comparison. We show the performance comparison in Table VI. xBS-GNN achieves up to 8.39× speedup compared with the DGL baseline. This is because xBS-GNN efficiently utilizes the multi-level memory along with our data placement policy optimized for GNN training, and also supports customized hardware kernels to process graph-structured data. On the other hand, GPU-based solutions do not have such fine-grained control over the hardware, leading to limited performance. On the MAG240M dataset, xBS-GNN achieves limited speedup over the DGL baseline because MAG240M has a long feature vector (768 dimensions), and the on-chip memory and HBM can only store a limited number of nodes; therefore, most of the data access happens in the FPGA DDR, which only has 38 GB/s of peak memory bandwidth.

D. Impact of optimizations

To evaluate the effectiveness of our optimizations, we compare the performance of our optimized designs with a FPGA baseline design. We show the speedup normalized to the FPGA baseline design in Figure 6. In the FPGA baseline design, the input graph is stored in the CPU main memory, and transferred to the FPGA in a mini-batch fashion during GNN training. We then apply our data placement policy (DPP) to store the input graph in the multi-level memory, which results in a 1.7×-7.2× speedup. The ogbn-products dataset shows a significant speedup because it is a relatively small dataset, and a high percentage of the frequently accessed nodes can be stored on-chip and in the HBM. We also apply feature quantization (without DPP) to the FPGA baseline design, and observe a 3.4×-3.9× speedup. The feature quantization effectively speeds up training on billion-scale graphs as data loading heavily dominates the overall training time. Finally, we

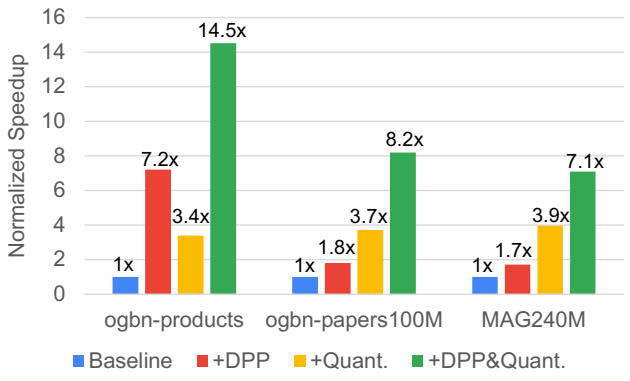

Fig. 6. Impact of optimizations

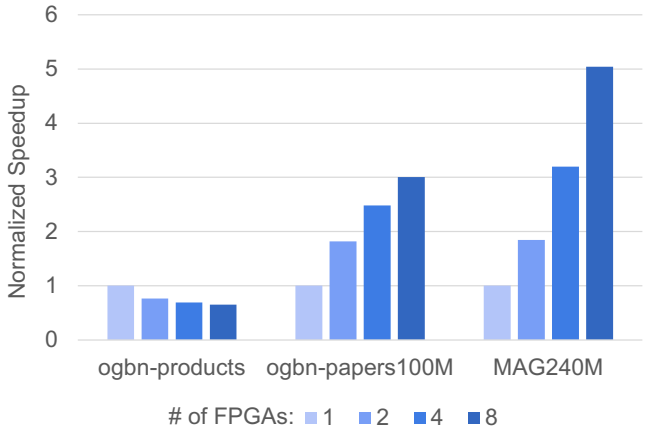

Fig. 7. xBS-GNN demonstrates good scalability to 8 FPGAs

apply both DPP and feature quantization, and achieve a 7.1×-14.5× speedup over the FPGA baseline design This combination shows a synergistic effect on billion-scale graphs, such as ogbn-papers100M and MAG240M. Specifically, assuming DPP and feature quantization contribute to the performance independently, applying both techniques theoretically results in a 6.6× speedup (1.8 times 3.7) on the ogbn-papers100M dataset; however, we observe an 8.2× speedup. MAG240M also shows a similar synergistic effect. This is because the feature quantization technique effectively complements the DPP: by compressing the node features into INT8, feature quantization allows more frequently accessed nodes to be placed on-chip, boosting the performance by up to 4.56× compared with applying DPP alone. In addition, it also increases hardware parallelism as modules implemented in INT8 requires less hardware resources than in FP32.

E. Scalability

We evaluate the scalability of xBS-GNN using the performance model proposed in HitGNN [44]. We assume the same platform setup as HitGNN, where the CPU is connected to multiple FPGAs via PCIe. The inter-FPGA communication is performed using a shared memory space allocated in the CPU main memory. To train on multiple FPGAs, we follow the algorithm proposed in P^3 [7], which partition the graph along the feature dimension, and assigns each partition to an FPGA for synchronous GNN training [45]. Figure 7 shows that xBS-GNN achieves good scalability on billion-scale graphs like ogbn-papers100M and MAG240M. This is primarily due to the increased memory resources available by deploying multiple FPGAs, which allows more frequently accessed nodes to be placed on-chip or in the HBM. Such an effect is more evident on MAG240M than ogbn-papers100M as MAG240M is larger in size, and can benefit more from the increased memory resources. Note that xBS-GNN does not scale on the ogbn-products dataset. The ogbn-products can be stored on-chip and in HBM on a single FPGA; therefore, it does not benefit from the increased memory resources provided by the multi-FPGA platform, and the performance degrades with more FPGAs deployed due to higher synchronization overhead. Nevertheless, for small datasets like ogbn-products, a single FPGA is sufficient to train.

V. Conclusion

In this work, we proposed xBS-GNN, an accelerator optimized for billion-scale GNN training. We proposed a novel data placement policy to store billion-scale graphs in the multi-level memory based on access frequency. xBS-GNN jointly exploited multiple optimizations to boost the training performance, including feature quantization, optimized hardware kernels, and a novel data placment policy along with a memory-efficient lookup table design. xBS-GNN achieved up to 8.39× speedup compared with state-of-the-art GNN library that runs on a RTX A5000 GPU. Compared with GNN accelerators running on multi-GPU platforms, xBS-GNN achieved up to 5.13× speedup using only a single FPGA. While we chose the GCN model for evaluation, xBS-GNN can be generalized to other models such as GraphSAGE [5] and GIN [46] by modifying the operators in the hardware kernels; since our kernels are implemented in HLS, such a modification can be done by simply updating a few lines of code. In the future, we plan to generalize xBS-GNN to support various GNN models, and run on distributed platforms to further improve performance.

Acknowledgment

This work is supported in part by the Semiconductor Research Corporation (SRC), and the U.S. National Science Foundation (NSF) under grants CCF-1919289/SPX-2333009 and OAC-2209563.

References

[1] D. S. Lopera, L. Servadei, G. N. Kiprit, S. Hazra, R. Wille, and W. Ecker, "A survey of graph neural networks for electronic design automation," in *Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2021.

[2] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM International Conference on Knowledge Discovery & Data Mining*, 2018.

[3] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: a comprehensive graph neural network platform," *Proceedings of the VLDB Endowment*, 2019.

[4] C.-I. Yang and Y.-P. Li, "Explainable uncertainty quantifications for deep learning-based molecular property prediction," *Journal of Cheminformatics*, 2023.

[5] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

[6] D. Zheng, X. Song, C. Yang, D. LaSalle, and G. Karypis, "Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs," in *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022.

[7] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[8] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "Pagraph: Scaling gnn training on large graphs via computation-aware caching," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020.

[9] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen, "Understanding and bridging the gaps in current gnn performance optimizations," in *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21, 2021.

[10] Y.-C. Lin and V. Prasanna, "Hyscale-gnn: A scalable hybrid gnn training system on single-node heterogeneous architecture," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023.

[11] B. Zhang, S. R. Kuppannagari, R. Kannan, and V. Prasanna, "Efficient neighbor-sampling-based gnn training on cpu-fpga heterogeneous platform," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021.

[12] H. Zeng and V. Prasanna, "Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020.

[13] Y.-C. Lin, B. Zhang, and V. Prasanna, "Hp-gnn: Generating high throughput gnn training implementation on cpu-fpga heterogeneous platform," in *International Symposium on Field-Programmable Gate Arrays*, 2022.

[14] W. Hu, M. Fey, H. Ren, M. Nakata, Y. Dong, and J. Leskovec, "Ogb-lsc: A large-scale challenge for machine learning on graphs," *arXiv preprint arXiv:2103.09430*, 2021.

[15] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," *CoRR*, vol. abs/1704.01212, 2017.

[16] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017.

[17] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graph-SAINT: Graph sampling based inductive learning method," in *International Conference on Learning Representations*, 2020.

[18] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[19] H. Zeng, M. Zhang, Y. Xia, A. Srivastava, A. Malevich, R. Kannan, V. Prasanna, L. Jin, and R. Chen, "Decoupling the depth and scope of graph neural networks," in *Advances in Neural Information Processing Systems*, 2021.

[20] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou, "Gnnlab: A factored system for sample-based gnn training over gpus," in *Proceedings of the 17th European Conference on Computer Systems*, 2022.

[21] Y.-C. Lin, G. Deng, and V. Prasanna, "A unified cpu-gpu protocol for gnn training," in *Proceedings of the 21st ACM International Conference on Computing Frontiers*, 2024.

[22] Y.-C. Lin, Y. Chen, S. Gobriel, N. Jain, G. K. Jha, and V. Prasanna, "Argo: An auto-tuning runtime system for scalable gnn training on multi-core processor," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024.

[23] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[24] B. Zhang, R. Kannan, and V. Prasanna, "Boostgcn: A framework for optimizing gcn inference on fpga," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021.

[25] S. Abi-Karam and C. Hao, "Gnnbuilder: An automated framework for generic graph neural network accelerator generation, simulation, and optimization," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 2023.

[26] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, "Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[27] X. Chen *et al.*, "Rubik: A hierarchical architecture for efficient graph neural network training," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[28] Y. Xu, Y. Wang, A. Zhou, W. Lin, and H. Xiong, "Deep neural network compression with single and multiple level quantization," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[29] Y. Guo, "A survey on methods and theories of quantized neural networks," *arXiv preprint arXiv:1808.04752*, 2018.

[30] S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei, "The era of 1-bit llms: All large language models are in 1.58 bits," *arXiv preprint arXiv:2402.17764*, 2024.

[31] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in Neural Information Processing Systems*, 2024.

[32] B. Feng, Y. Wang, X. Li, S. Yang, X. Peng, and Y. 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)*, 2020.

[33] Y. Ma, P. Gong, J. Yi, Z. Yao, C. Li, Y. He, and F. Yan, "Bifeat: Supercharge gnn training via graph feature quantization," in *arXiv preprint arXiv:2207.14696*, 2023.

[34] M. Ding, K. Kong, J. Li, C. Zhu, J. Dickerson, F. Huang, and T. Goldstein, "Vq-gnn: A universal framework to scale up graph neural networks using vector quantization," in *Advances in Neural Information Processing Systems*, 2021.

[35] S. A. Tailor, J. Fernandez-Marques, and N. D. Lane, "Degree-quant: Quantization-aware training for graph neural networks," in *International Conference on Learning Representations*, 2021.

[36] D. Zheng *et al.*, "Distdgl: distributed graph neural network training for billion-scale graphs," in *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2020.

[37] J. Zhou, J. Wu, Y. Gao, Y. Ding, C. Tao, B. Li, F. Tu, K.-T. Cheng, H. K.-H. So, and N. Wong, "Dybit: Dynamic bit-precision numbers for efficient quantized neural network inference," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[38] Y. Wong, Z. Dong, and W. Zhang, "Low bitwidth cnn accelerator on fpga using winograd and block floating point arithmetic," in *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2021.

[39] Y. Fu, E. Wu, and A. Sirasa, "8-bit dot-product acceleration," in *White Paper: UltraScale and UltraScale+ FPGA*, 2017.

[40] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "Thundergp: Hls-based graph processing framework on fpgas," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021.

[41] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[42] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.

[43] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[44] Y.-C. Lin, B. Zhang, and V. Prasanna, "Hitgnn: High-throughput gnn training framework on cpu+multi-fpga heterogeneous platform," in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2024.

[45] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," in *International Conference on Learning Representations Workshop*, 2016.

[46] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2019.