Efficient Neighbor-Sampling-based GNN Training on CPU-FPGA Heterogeneous Platform

Bingyi Zhang*, Sanmukh R. Kuppannagari*, Rajgopal Kannan†, Viktor Prasanna*
*University of Southern California, Los Angeles, USA
†US Army Research Lab, Los Angeles, USA
Email: bingyizh@usc.edu, kuppanna@usc.edu, rajgopal.kannan.civ@mail.mil, prasanna@usc.edu

***Abstract*—Graph neural networks (GNNs) have become increasingly important in many real-world applications. However, training GNN on large-scale real-world graphs is still challenging. Many sampling-based GNN training algorithms have been proposed to facilitate the mini-batch style training process. The well-known Neighbor-Sampling-base (NS) GNN training algorithms, such as GraphSAGE, have shown great advantages in terms of accuracy, generalization, and scalability on large-scale graphs. Nevertheless, efficient hardware acceleration for such algorithms has not been systematically studied.**

In this paper, we conduct an experimental study to understand the computational characteristics of NS GNN training. The evaluation results show that *neighbor sampling* and *feature aggregation* take the majority of the execution time due to the irregular memory accesses and extensive memory traffic. Then, we propose a system design for NS GNN training by exploiting the CPU-FPGA heterogeneous platform. We develop an optimized parallel neighbor sampling implementation and an efficient FPGA accelerator to enable high-throughput GNN training. We propose the neighbor sharing and task pipelining techniques to improve the training throughput. We implement a prototype system on an FPGA-equipped server. The evaluation results demonstrate that our CPU-FPGA design achieves $12-21\times$ speedup than CPU-only platform and $0.4-3.2\times$ speedup than CPU-GPU platform. Moreover, our FPGA accelerator are $2.3\times$ more energy efficient than the GPU board.

***Index Terms*—Graph Neural Network, Training, Graph Sampling, CPU-FPGA Heterogeneous Platform**

I. Introduction

Graph Neural Networks (GNNs) have become a revolutionary machine learning technique for many real-world applications [1], [2] where the underlying data can be modeled as a graph. GNNs were first proposed in [3] where the authors use the entire graph as input to train the GNN model in each training iteration. This technique is called full-graph GNN training. However, it has poor scalability when graph sizes are large, as it the case in most real world applications. Moreover, full-graph GNN training can potentially lead to accuracy loss due to overfitting [4].

To enable training of GNN models on large-scale graphs, many sampling-based training methods [5], [6], [7] have been proposed, which perform GNN training in mini-batches. Among these methods, neighbor-sampling-based (NS) method (e.g. GraphSAGE [5], PinSAGE [2]) have shown excellent performance (in terms of accuracy) on large scale graph for a variety tasks as evident from the leaderboard of the Open graph benchmark [4]. In these algorithms, in each training iteration, a batch of target vertices is selected from the input graph. The neighbor sampler recursively samples the neighbors, starting from the target vertices, according to a certain probability distribution. Then, forward propagation and backpropagation are performed on the sampled mini-batch. The GNN layer weights are updated based on the calculated gradients. From an algorithm perspective, the neighbor sampling technique works as a drop-out mechanism [8] to resolve overfitting of the model. From a computation perspective, by adopting the mini-batch training, GNN training can be deployed on accelerators such as GPUs with limited global memory size.

While NS GNN models are widely used in real-world applications, the hardware acceleration for the training of these models has not been systematically studied. In this paper, we carry out an experimental study to understand the computational characteristics of the training of NS GNNs. The evaluation results show that the *neighbor sampling* and *feature aggregation* are the major bottlenecks that take 81%-94% of the total execution time.

In this paper, we propose a CPU-FPGA system to accelerate NS GNN training. In the proposed system consists of a parallel neighbor sampling algorithm running on the host processor coupled with an optimized FPGA accelerator to execute GNN operations. To achieve high throughput, We propose several optimizations to improve the memory performance and computation efficiency.

Our main contributions are summarized as follow:

- We perform a detailed experimental study to understand the computational characteristics of the NS GNN training methods.
- We propose a system targeting CPU-FPGA heterogeneous platform to accelerate NS GNN training.
- We propose several optimizations, such as neighbor sharing and task pipelining, to improve the computation efficiency of NS GNN training.
- We conduct experiments on an FPGA-equipped server to evaluate the proposed system. The evaluation results show that our proposed CPU-FPGA implementation achieves $12-21\times$ speedup than CPU-only platform and $0.4-3.2\times$ speedup than CPU-GPU platform.

II. Background

We define the required notations in Table I. Graph Neural Networks (GNN) [3] are proposed for representation learning

TABLE I
GNN NOTATIONS

Notation	Description	Notation	Description
$\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X^0})$	input graph	v_i	i^{th} vertex
$\mathcal{V}$	set of vertices	e_{ij}	edge from v_i to v_j
$\mathcal{E}$	set of edges	L	number of GNN layers
N	number of vertices	nnz	number of edges
$\boldsymbol{X^0} \in \mathbb{R}^{N \times f_0}$	feature matrix of $\mathcal{V}$	$\boldsymbol{X^l} \in \mathbb{R}^{N \times f_l}$	feature matrix of layer l
h_i^l	feature vector of v_i	$\mathcal{N}(i)$	neighbors of v_i

on graphs. GNNs learn to generate low-dimensional vector representation that capture the structural information (e.g. edges $\mathcal{E}$) and vertex features $\boldsymbol{X}^0$ of the graphs. The learned vector representation can be used for many downstream tasks, such as node classification [5], link prediction [9], etc. To overcome the poor scalability of full-graph training, many sampling-based GNN training algorithms have been proposed. Broadly, the sampling techniques for GNN training can be divided into two categories: (1) neighbor sampling , and (2) subgraph sampling. In neighbor-sampling-based GNN training [5], [2], the mini-batch sampler recursively samples the k-hop neighbors for the target vertices. In each training iteration, the GNN operation is performed on a batch of target vertices within sampled k-hop neighbors. In subgraph-sampling-based GNN training, the sampler samples a mini-batch subgraph through graph partition [10], edge-based sampling [6] or random walk [6]. The GNN operation is performed within the sampled subgraph. The evaluation in [4] shows that there is no one-fit-all sampling technique for various domains. GraphACT [11] accelerates subgraph-sampling-based GNN training on CPU-FPGA heterogeneous platform. Some previous work [12], [13] focus on GNN inference. No existing works study the hardware acceleration for neighbor-sampling-based GNN training.

III. NEIGHBOR-SAMPLING-BASED GNN TRAINING

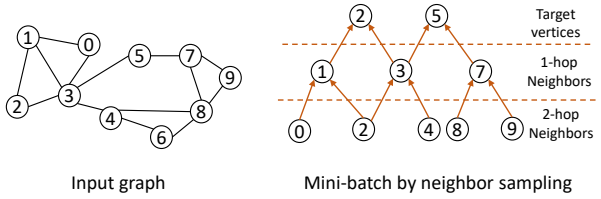


Fig. 1. A toy example of neighbor sampling.

Neighbor sampling is performed by recursively sampling the k-hop neighbors from a batch of target vertices. The basic sample function is denoted as:

$$\mathcal{N}_s(v) = \text{SAMPLE}(\mathcal{N}(v), \mathbb{P}, d), \tag{1}$$

where $\mathcal{N}_s(v)$ denotes the set of sampled neighbors for vertex v. $\mathcal{N}(v)$ denotes the set of neighbors of vertex v. $\mathbb{P}$ is the probability distribution function over vertices $u \in \mathcal{N}(v)$, which denotes the probability of sampling a vertex u which is a neighbor of v. d is the neighbor sampling budget which denotes the number of neighbors of each vertex v to be included in the sampled set. Figure 1 shows an example of recursively sampling the 2-hop neighbors for target vertices $\{2, 5\}$.

The overall neighbor-sampling-based GNN training algorithm is shown in Algorithm 1. In each training iteration, a set of target vertices V^T is selected. Lines 7-14 show the recursive neighbor sampling process for GNN layer construction $\{B_l : 0 \leqslant l \leqslant L\}$ based on V^T. B_l denotes the set of sampled l-hop neighbors for V^T. After the sampling step, GNN forward propagation and backpropagation operate within the sampled L-hop neighbors $\{B_l : 0 \leqslant l \leqslant L\}$. In forward propagation (Line 16-21), GNN-layer operations consist of two major computation kernels – *feature aggregation* and *feature update*. *Feature aggregation* follows the message-passing paradigm, and each vertex aggregates features from the sampled neighbors. In *feature update phase*, the aggregated feature vectors are transformed by a Multi-Layer Perceptron with non-linear activation function. Backpropagation has a similar computation pattern as forward propagation.

Algorithm 1 Neighbor-sampling-based mini-batch GNN training algorithm

Input: Input graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X}^0)$, L: number of GNN layers, SAMPLE$^l()$: neighbor sampler for each layer with budget d^l $(1 \leqslant l \leqslant L)$ and probability distribution $\mathbb{P}$. s: size of a mini-batch. $\{W^l : 1 \leqslant l \leqslant L\}$: GNN layer weights.
Output: Trained GNN model with GNN-layer operation $\{\text{aggregate}^l(), \text{update}^l() : 1 \leqslant l \leqslant L\}$.

1: **repeat**
2: Vertex-shuffling($\mathcal{V}$)
3: { % Training iteration %}
4: **for** $i \leftarrow 0$ to $\frac{N}{s} - 1$ **do**
5: $V^T = \{v_j : s * i \leqslant j \leqslant s * (i+1) - 1, v_j \in \mathcal{V}\}$
6: $B_L \leftarrow V^T$
7: { % Recursive neighbor sampling %}
8: **for** $l \leftarrow L$ to 1 **do**
9: $B_{l-1} \leftarrow B_l$
10: **for** $u \in B_{l-1}$ **do**
11: $\mathcal{N}_s^{l-1}(u) = \text{SAMPLE}^l(\mathcal{N}(u), \mathbb{P}, d^l)$
12: $B_{l-1} \leftarrow B_{l-1} \cup \mathcal{N}_s^{l-1}(u)$
13: **end for**
14: **end for**
15: { % Forward propagation %}
16: **for** $l \leftarrow 1$ to L **do**
17: **for** each $u \in B_l$ **do**
18: $z_u^l = \text{aggregate}^l(h_v^{l-1} : v \in \mathcal{N}_s^{l-1}(u))$
19: $h_u^l = \text{update}^l(z_u^l, h_u^{l-1}, W^l)$
20: **end for**
21: **end for**
22: **BackPropogation**()
23: **WeightUpdate**()
24: **end for**
25: **until** convergence

IV. PERFORMANCE PROFILING

To study the computational characteristics of the neighbor-sampling-based GNN training, we conduct experiments on the CPU platform and CPU-GPU platform. The CPU platform has an Intel i9-9900K processor with 8 cores and 16 threads.

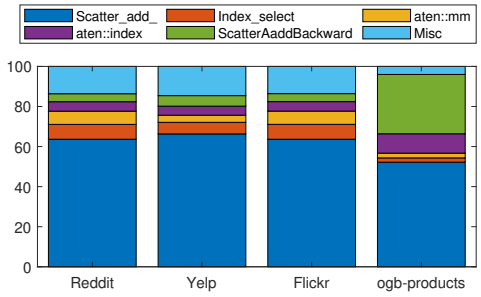


Fig. 2. The breakdown execution time on the CPU platform

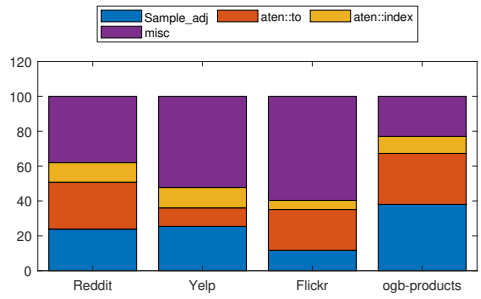


Fig. 3. The breakdown execution time on the CPU-GPU platform

The CPU-GPU platform has the same processor as the CPU platform. It is equipped with an Nvidia RTX 3070 GPU. We run the representative NS training algorithm GraphSAGE [5] using Pytorch Geometric[1]. Four widely used datasets – Reddit [5], Yelp [6], Flickr [6], ogb-products [4] – are used for evaluation. The batch size is set to 1024.

The performance breakdowns of CPU platform and CPU-GPU platform are shown in Figure 2 and Figure 3, respectively. On the CPU platform, feature aggregation (Scatter_add_, Index_select, ScatterAddBackward) dominates the execution time. This is because feature aggregation is communication-intensive, and CPU platform only provides limited memory bandwidth (<19.2 GB/s). Thus, feature aggregation becomes the major bottleneck on the CPU platform. On the CPU-GPU platform, neighbor sampling is performed on the host CPU while the GNN operations are executed on GPU. The performance breakdown shows that neighbor sampling (Sample_adj) and data copy (aten::to) take most of the execution time. Because of the underlying irregular graph structure, neighbor sampling leads to random memory accesses, thereby leading to degraded performance. The sampled mini-batches are copied from host memory to GPU global memory through PCIe connection, which usually has limited bandwidth. Therefore, neighbor sampling and data copy become the two major bottlenecks on the CPU-GPU platform.

From the profiling results, we summarize two key takeaways:

- In neighbor-sampling-based GNN training, neighbor sampling, and feature aggregation are the major bottlenecks, due to the random memory accesses and intensive memory traffic.
- On a heterogeneous platform such as CPU-GPU, the overhead of data movement between the host processor and the accelerator is significant for neighbor-sampling-based GNN training.

V. System Design

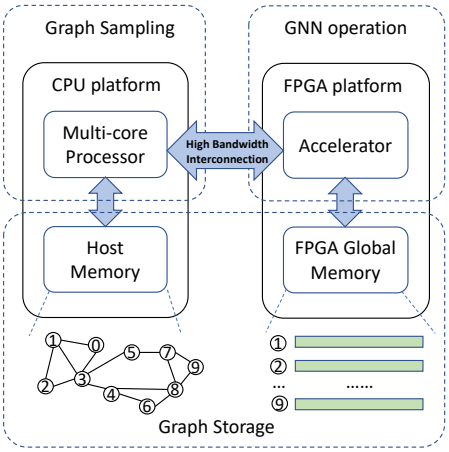


Fig. 4. The proposed system design

Given the challenges, we propose a system design that exploits the CPU-FPGA heterogeneous platform for neighbor-sampling-based GNN training. The proposed system is composed of three components that perform *graph storage*, *graph sampling* and *GNN operation*. The graph sampling is performed on the host processor as it requires complex data structure and numeric operations, which are suitable to be executed on the general purpose processor. On the FPGA board, we build a customized accelerator to exploit the fine-grained computation parallelism in GNN operations. The system memory consists of host memory and FPGA global memory. Host memory stores the graph structural information to facilitate graph sampling on the multi-core processor, while the vertex features are stored in the FPGA global memory for GNN operations. The CPU platform and FPGA platform are connected through a high bandwidth interconnection for efficient data movement. For example, many high-speed interconnection technologies, such as Intel UPI, CXL are integrated into the state-of-the-art FPGAs.

We summarize the benefits of using the FPGA platform for GNN training: (1) FPGA platform is more energy efficient than GPU platform in data center, thus it can potentially save the energy cost for GNN training; (2) FPGA platform allows fine grained customization that enables implementation of application dependent memory access pattern and massive computation parallelism; (3) FPGA platform can support large local DRAM size. For example, a commercial FPGA board[2] has up to 260 GB local DRAM. Therefore, we can store the entire vertex features in the FPGA local DRAM. This can reduce the overhead of data copy.

[1]https://github.com/rusty1s/pytorch_geometric

[2]https://gidel.com/acceleration-platforms/

A. Graph Sampling

Vertex ID	Adjacency List
0	1 3
1	0 2 3
2	1 3
3	0 1 2 4 5
4	3 6 8
5	3 7
6	4 8
7	5 8 9
8	4 6 7 9
9	7 8

Adjacency Table

Vertex ID	Pruned Adjacency List (L=3)
0	1 1 3
1	0 2 3
2	1 3 3
3	0 4 5
4	3 6 8
5	3 3 7
6	4 8 8
7	5 8 9
8	4 6 9
9	7 7 8

Pruned Adjacency Table

Fig. 5. The table on the left is the adjacency table of input graph in Figure 1. The table on the right is the pruned adjancy table.

Graph sampling is performed by recursively traversing the neighbors starting from a batch of target vertices. The neighbor sampling process is similar to the multi-source Breath-first search (BFS). Due to the irregular graph structure, fetching the neighbors introduces random memory accesses, leading to long memory latencies. While the random memory access in BFS can be eliminated by the two-phase edge-centric graph processing paradigm [14], [15], it cannot be easily eliminated in neighbor sampling where the SAMPLE function is usually randomized. Therefore, we exploit two techniques to improve the efficiency of graph sampling: *Look-up Table based neighbor fetching* and *multi-threading*.

Look-up Table based neighbor fetching: We map the neighbor sampling process into the Look-up Table operation. The graph structure can be represented using the adjacency table, with each vertex having an adjacency list. The adjacency list contains the neighbors of the vertex. However, each vertex has an adjacency list of different lengths, making the address of the adjacency list unpredictable. Obtaining the address of the adjacency list requires extra random memory access. Therefore, we propose to prune the adjacency list. We set a parameter $prunedLength$, so that the adjacency list of each vertex is pruned to be length of $prunedLength$. Using pruned adjacency list, we can infer the address of the adjacency list by $vertex\_ID \times prunedLength$, which is represented as the look-up table operation. The pruned adjacency list is built by random sampling with replacement from the original adjacency list. Note that the pruned adjacency table is built for each training epoch. Each vertex has an equal probability of being sampled into the pruned adjacency list, and the overhead of building pruned adjacency table is amortized within the training epoch. The evaluation results show that such a pruning strategy has a negligible effect on the training accuracy.

Multi-threading: While the Look-up Table based neighbor fetching strategy can eliminate the latency of obtaining the address of vertex adjacency list, there is still large latency of fetching the adjacency list due to the random memory accesses. To hide the memory latency, we exploit the multi-threading technique such that the samplers can work in parallel to perform neighbor sampling. When one sampler is waiting for neighbor fetching, the other sampler is scheduled to execute sampling. Each sampler samples a mini-batch independently, and the sampled mini-batch is added to the task pool. Note that using multiple threads to sample a single mini-batch will introduce large overhead for synchronization to avoid duplicate vertices, while our parallel sampling is lock-free.

B. GNN operation and accelerator design

To efficiently execute GNN operations, we develop a customized accelerator on the FPGA board as shown in Figure 6. The FPGA local DRAM stores the vertex features and GNN layer weights. The accelerator design consists of three hardware modules – Feature Aggregation Module (FAM), Feature Transformation Module (FTM), Weight Update Module (WUM). The memory controller handles the data transmissions between FPGA local DRAM and hardware modules.

Feature Aggregation Module: In the feature aggregation phase, vertices (destination vertices) aggregate features vectors from the sampled neighbors (source vertices). We apply the scatter-gather paradigm to execute feature aggregation in FAM. In scatter phase, the edge weights are applied to the source vertex features to generate messages in Message Generators. Then, messages are routed to the destination vertices through the message shuffling network (MSN). We exploit the butterfly network [16] for the message shuffling. The Message Aggregators read the destination vertex from the vertex buffer and apply the messages to the destination vertices. FAMs are fully pipelined to increase the computation throughput. RAW Resolvers resolve the potential *read-after-write* (RAW) data hazards in Message Aggregators. To increase the computation parallelism, Message Generators and Message Aggregators are organized in SIMD fashion, so that they can process multiple vertex features in each clock cycle. Note that in GNN backpropagation, the vertices propagate gradients to the sampled neighbors, which has a similar computation pattern as feature aggregation. Therefore, the scatter-gather paradigm can also be applied in backpropagation.

Feature Transformation Module: In the feature transformation phase, the vertex feature vectors are transformed by the GNN layer weight. Several vertex feature vectors can be processed in parallel to increase the computation parallelism. We exploit the 2-D systolic array to execute the multiplication between feature vectors and weight matrix.

Weight Update Module: After getting the gradients, Weight Update Module calculates the gradients for the GNN layer weights and apply the gradients to the weights according to the selected weight updating strategy.

C. Optimizations

Neighbor Sharing: As shown in section IV, feature aggregation is a major bottleneck in GNN training due to the large memory traffic. We notice that in neighbor sampling, target vertices may share the common neighbors. So, we propose to use the neighbor sharing technique by reusing the common neighbors in the FPGA on-chip memory. Using Figure 1 as the example, when propagating information from 2-hop neighbors

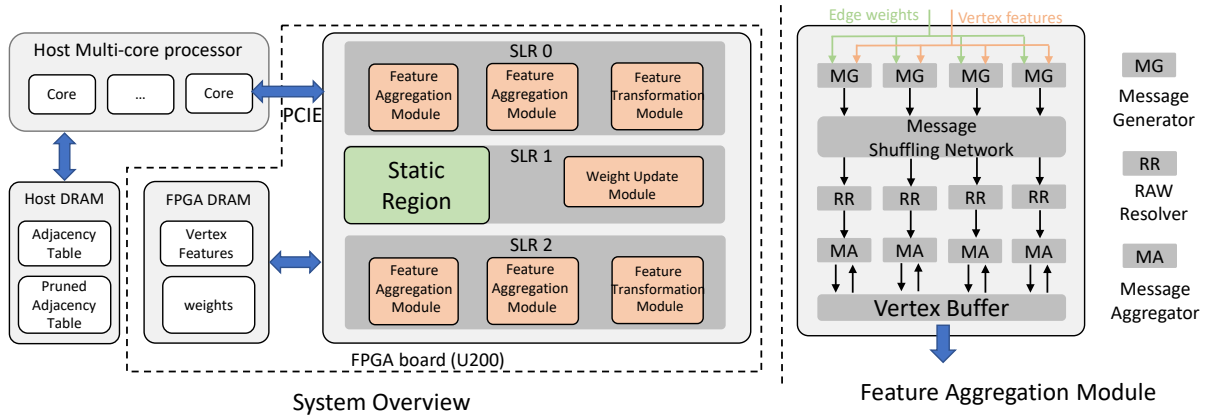


Fig. 6. The figure on the left demonstrate the system overview. The figure on the right shows the details of the feature aggregation module.

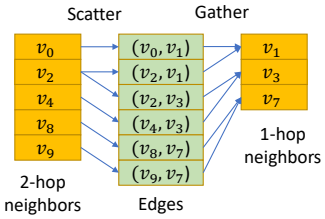


Fig. 7. Data structure

to 1-hop neighbors, vertex 2 is shared by vertices 1 and 3. Therefore, by caching vertex 2 in the on-chip memory, vertex 2 can be used twice. To enable neighbor sharing, we sort the vertices by their indices and sort the edges (stored in COO format, $< source, destination >$) by the indices of the source. The data structure is shown in Figure 7. The feature aggregation module sequentially reads the source vertices and edges to generate the messages. Therefore, consecutive edges could share the same source vertices. Then, the messages are applied to the destination vertices, which are stored in the Vertex Buffer.

Tasking pipelining: To improve the training throughput, we pipeline the three tasks – graph sampling, data transmission and GNN operations. Multiple parallel samplers are sampling the mini-batches and transfer the mini-batches to the task pool, which resides in the FPGA local memory. The data transmission is through the high-bandwidth interconnection between the host platform and FPGA platform. The FPGA accelerators load the mini-batches from the task pool for GNN operations.

VI. Experimental Evaluation

A. Implementation

We deploy the proposed design on an FPGA-equipped server for evaluation. The host processor of the server is Intel Xeon Gold 5120 CPU. A Xilinx Alveo U200 FPGA board is connected to the host processor through PCIe. Alveo U200 has three Super Logic Regions (SLRs), and the Xilinx FPGA Shell is deployed in SLR1. The effective PCIe bandwidth between the host processor and FPGA board is nearly 11.9 GB/s. We compare our implementation with two baseline implementations:

- **CPU-only**: All the tasks are performed on the host processor.
- **CPU+GPU**: The host processor is connected with a high-end GPU – Titan XP. The host processor performs the graph sampling and the GPU performs the GNN operations.

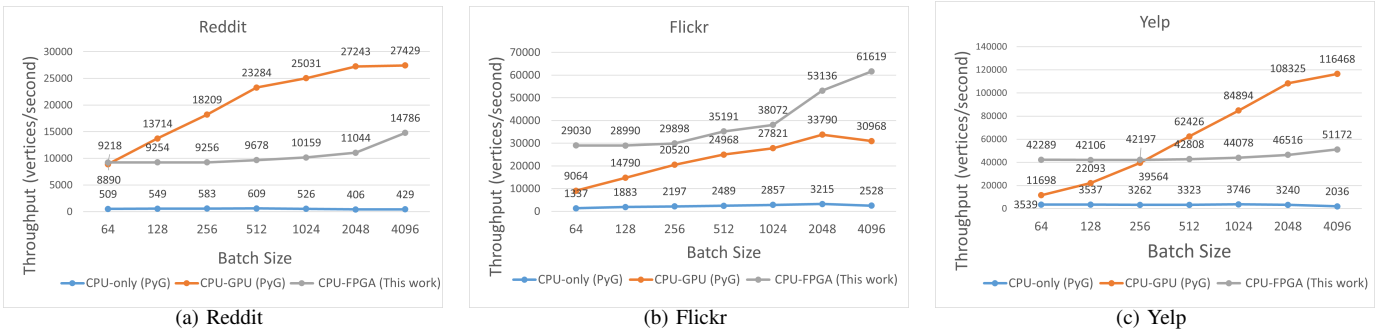
The three implementations use the same host processor. For the CPU-only and CPU+GPU platforms, we run the GNN training using the state-of-the-art graph learning framework – Pytorch Geometric (PyG). For the proposed system, we run our parallel graph sampling implementation (Section V-A) on the host processor, and we run the GNN operations on our proposed FPGA accelerator. For the three implementations, we train the 2-layer GraphSAGE model with parameters used in [5]. The hidden dimension is set to 128. The sampling budgets (d^1, d^2) for layers 1 and 2 are set to $(10, 25)$. All the three implementations use the 16 parallel neighbor samplers.

B. FPGA Resource Utilization

In each of SLR0 and SLR2, we deploy two Feature Aggregation Modules and one Feature Transformation Module. Each Feature Aggregation Module has two Message Generators, and each Message Generator can process 16 vertex features per clock cycle. The Feature Transformation Module is implemented using a systolic array of size 16×16. The Weight Update Module is implemented using a multiply-accumulate array of size 8×8 to accumulate the gradients and update the layer weights. The design is developed using Xilinx High-Level Synthesis (HLS). The FPGA accelerator uses 652K LUTs, 1592 BRAMs, 512 UltraRAMs, and 3456 DSPs. The running frequency is 270 MHz. The reported resource utilization and frequency are obtained from Xilinx Vitis 2020.1 after Place & Route.

C. Comparison of Training Throughput

For the comparison of throughput, we run the GNN training using three widely used datasets *Reddit*, *Filckr* and *Yelp* as shown in Table II. We run the GNN training for 10 epochs



(a) Reddit (b) Flickr (c) Yelp

Fig. 8. The comparison of performance under various platforms

TABLE II
DATASET STATISTICS

Dataset	Vertices	Edges	Features	Classes	Degree
Flickr [6]	89250	899756	500	7	10
Reddit [5]	232965	116069191	602	41	50
Yelp [6]	716847	6977410	300	100	10

to measure the average training throughput. The training throughput is calculated by:

$$\text{Throughput} = \frac{(\#\text{ of vertices}) \times (\#\text{ of epochs})}{\text{Execution Time}} \quad (2)$$

For the CPU-only implementation, both the graph structural information and the vertex features are stored in the Host DRAM. For the CPU-FPGA and CPU-GPU implementation, the graph structural information is stored in the Host DRAM, and the vertex features are stored in the GPU/FPGA local DRAM. We measure the training throughput under different batch sizes, 64, 128, 256, 512, 1024, 2048, 4096. The evaluation results are shown in Figure 8.

Compared with the CPU-only implementation, our CPU-FPGA implementation achieves $18-19\times$, $13-21\times$, $12\times$ speedup on Reddit, Flickr, Yelp, respectively. The achieved speedup is due to more computation resources introduced by the FPGA accelerator. Compared with the CPU-GPU implementation, our CPU-FPGA implementation achieves $0.4-3.2\times$ speedup. CPU-FPGA achieves better performance than CPU-GPU when batch size is small. When the batch size is small, the memory bandwidth and the computation power of GPU are not saturated. When the batch size is larger than 2048, GPU becomes saturated, and our CPU-FPGA implementation can still achieve 40% performance of CPU-GPU implementation in the worst case. Note that the FPGA board (Alveo U200) only has 77 GB/s memory bandwidth and the GPU board (Titan X) has 547 GB/s memory bandwidth. Moreover, our implemented FPGA accelerator has the peak performance of 0.241 TFLOPS, and the GPU board has the peak performance of 9.3 TFLOPS. Therefore, our CPU-FPGA implementation has higher computation efficiency than CPU-GPU implementation. The training throughput of our CPU-FPGA implementation increases with larger batch size. This is because when the batch size becomes larger, the neighbor sharing becomes more prominent leading to increased data reuse. Note that CPU-FPGA implementation has more speedup on Flickr than on Reddit and Yelp. Because Flickr has small average degree and small number of vertices, leading to more neighbor sharing for FPGA implementation. The CPU-GPU implementations does not exploit the neighbor sharing.

To compare energy efficiency of GPU and our proposed FPGA accelerator, we estimate the power consumption of GPU using the thermal design power (TDP). The power consumption of the FPGA board is measured using Xilinx XRT. The TDP of GPU is 250W, while the FPGA board has an average power consumption of 42W, including the power consumption of PCIe. The energy effciency is caculated by: Energy Efficiency $= \frac{\text{Throughput}}{\text{Power}}$. The evaluation result shows the proposed FPGA accelerator is $2.3\times$ more energy efficient than the GPU board.

VII. CONCLUSION

In this paper, we provide an optimized implementation for the neighbor-sampling-based GNN training on CPU-FPGA heterogeneous platform. We propose a parallel neighbor sampling algorithm to improve the efficiency of graph sampling and we develop an optimized GNN accelerator for the high-throughput and energy-efficient GNN operations. The evaluation results show that our CPU-FPGA implementation achieves $12-21\times$ and $0.4-3.2\times$ speedup than the CPU-only and CPU-GPU implementations, respectively. In the future, we intend to support more GNN training algorithms, such as subgraph-based GNN training algorithms.

VIII. ACKNOWLEDGEMENT

This work has been sponsored by the U.S. National Science Foundation under grant numbers OAC-1911229, CNS-2009057 and CCF-1919289. Equipment and support by Xilinx are greatly appreciated.

REFERENCES

[1] H. Yang, "Aligraph: A comprehensive graph neural network platform," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 3165–3166.

[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 SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 974–983.

[3] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[4] 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.

[5] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *arXiv preprint arXiv:1706.02216*, 2017.

[6] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graphsaint: Graph sampling based inductive learning method," *arXiv preprint arXiv:1907.04931*, 2019.

[7] J. Chen, T. Ma, and C. Xiao, "Fastgcn: fast learning with graph convolutional networks via importance sampling," *arXiv preprint arXiv:1801.10247*, 2018.

[8] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[9] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," *Advances in Neural Information Processing Systems*, vol. 31, pp. 5165–5175, 2018.

[10] 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 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 257–266.

[11] 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, pp. 255–265.

[12] B. Zhang, H. Zeng, and V. Prasanna, "Hardware acceleration of large scale gcn inference," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2020, pp. 61–68.

[13] 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, pp. 29–39.

[14] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "Hitgraph: High-throughput graph processing framework on fpga," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2249–2264, 2019.

[15] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 347–358.

[16] Y.-k. Choi, Y. Chi, W. Qiao, N. Samardzic, and J. Cong, "Hbm connect: High-performance hls interconnect for fpga hbm," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 116–126.