Accelerating GNN Training on CPU+Multi-FPGA Heterogeneous Platform

Yi-Chien Lin(✉), Bingyi Zhang, and Viktor Prasanna

University of Southern California, Los Angeles, USA
{yichienl,bingyizh,prasanna}@usc.edu

Abstract. Training Graph Neural Networks (GNNs) has become time consuming as the graphs grow larger. Thus, many works have been proposed to accelerate GNN training on multi-GPU platforms. Though GPUs feature high computation power, training GNNs on GPU suffers from low resource utilization. We propose to accelerate GNN training on a CPU+Multi-FPGA heterogeneous platform. By utilizing the customizable hardware resources on the FPGAs, we instantiate multiple hardware kernels with optimized data access pattern and memory organization. The optimized hardware kernels can efficiently access graph-structured data and thus achieve high training performance. However, training GNN with multiple FPGAs also leads to high FPGA-to-FPGA communication overhead and workload imbalance. We develop optimized graph partitioning techniques to minimize FPGA-to-FPGA data communication, and develop a task scheduler to balance the workload among the FPGAs. Compared with the state-of-the-art GNN training implementation on a multi-GPU platform, our work achieves up to 24.7× bandwidth efficiency; this superior efficiency enables our work to achieve up to 3.88× speedup and 7.18× energy efficiency using much less compute power and memory bandwidth than GPUs.

Keywords: Graph neural networks · CPU+Multi-FPGA

1 Introduction

Graph Neural Networks (GNNs) have facilitated many applications such as social recommendation system [21], molecular property prediction [9], and traffic prediction [12], etc. Despite the usefulness of GNNs, training a GNN model on a large-scale graph using a single GPU is time-consuming. Thus, there has been an increasing interest in using multi-GPU platforms [7,18] to accelerate GNN training. Although these works accelerate GNN training using multiple GPUs, some challenges remain: (1) inefficiency: GNN training underutilizes the available resources because traditional cache policies fail to capture the data access pattern in GNN training, resulting in high cache miss rate [11,20]. In addition, each data element goes through multi-level caches before being computed, which incurs high latency. Due to the aforementioned reasons, most of the training time

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
P. Navaux et al. (Eds.): CARLA 2022, CCIS 1660, pp. 16–30, 2022.
https://doi.org/10.1007/978-3-031-23821-5_2

is spent on reading and writing data from/to the GPU global memory, instead of the actual computation; (2) power consumption: though GPU features superior peak performance, it also comes with high power consumption. Power consumption is an essential consideration for cloud service providers like Amazon Web Service (AWS) and Microsoft Azure since it directly relates to the operating cost of the data centers.

Recently, Field Programmable Gate Array (FPGA) has emerged as a popular platform to accelerate GNN inference [16,25] and training [17,22]. This is because FPGAs are highly customizable; this allows developers to customize their hardware kernels, datapath, and memory hierarchy. In contrast, in CPUs and GPUs, the datapath, memory controller and memory hierarchy are all fixed. Utilizing the customized hardware designs, previous works [16,17,22,25] achieve high performance and energy-efficient GNN computations on a single FPGA platform. Cloud platforms like AWS F1 [1], Azure NP-series [2], and Intel DevCloud [3] are all equipped with FPGAs, making FPGAs easily accessible to researchers and developers.

Motivated by the challenges of training GNN on GPU platforms and the emergence of FPGAs, we propose to accelerate GNN training on a CPU+Multi-FPGA heterogeneous platform; such a platform consists of a multi-core CPU processor, connected to multiple FPGAs. We utilize the flexibility of CPU to perform control-intensive tasks such as graph preprocessing, mini-batch sampling and task scheduling. We exploit customizable hardware resources of FPGAs to develop kernels with optimized memory organization and data access pattern to reduce the communication overhead during GNN training. In addition to efficient data access, training GNNs with application-specific architecture on FPGAs allows us to achieve superior energy efficiency. Though a CPU+Multi-FPGA heterogeneous platform provides more hardware resources and memory bandwidth than a single FPGA platform, it is challenging to achieve a scalable speedup due to the complex data dependency of graph-structured data. In particular, during GNN training, the input graph is partitioned and distributed to each FPGA and trained in parallel. However, a straightforward graph partitioning would lead to significant FPGA-to-FPGA communication overhead [7] since each FPGA may need to read significant amount of data from other FPGA local DDR memory. To overcome this issue, we use METIS [13] to partition the input graph; METIS graph partitioning can minimize edge-cut between graph partitions and thus minimize data communication among the FPGAs. However, since each graph partition contains different number of vertices and edges, the workload of each partition is different. Thus, we develop a task scheduler to handle the workload imbalance among the FPGAs. Though we exploit multi-level parallelism and various optimization techniques, none of them alter the GNN training algorithm; thus, we achieve the same training accuracy and convergence rate as in training on a multi-GPU platform. We summarize our contributions as follows:

- We accelerate GNN training on a CPU+Multi-FPGA heterogeneous platform. We demonstrate the acceleration of GNN training using two well-known GNN models on three widely-used datasets.

- We develop hardware kernels with optimized memory organization and data access pattern, which reduce the data access overhead in GNN training.
- We develop several optimizations, including: (1) graph partitioning, and (2) workload balancing to improve the training performance on our target platform.
- Compared with a state-of-the-art GNN training framework on a multi-GPU platform, our implementation on a CPU+multi-FPGA platform achieves up to 24.7× bandwidth efficiency, 3.88× speedup, and 7.18× energy efficiency.

2 Background

2.1 GNN Models

Given an input graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X})$, where $\mathcal{V}$, $\mathcal{E}$, and $\boldsymbol{X}$ is the vertices, edges, and vertex features of the graph, a GNN model is specified by:

- L: number of layers.
- $\mathcal{V}^t$: a set of target vertices to be inferred.
- f^l: hidden dimension in layer l $(1 \leqslant l \leqslant L)$.
- A mechanism of constructing mini-batches, including:
  - The mechanism to construct $\mathcal{V}^l$: the set of vertices in layer l $(0 \leqslant l \leqslant L)$. $|\mathcal{V}^l|$ denotes the number of vertices in layer l. Moreover, $\mathcal{V}^L = \mathcal{V}^t$.
  - The mechanism to construct $\boldsymbol{A}^l \in \mathbb{R}^{|\mathcal{V}^{l-1}| \times |\mathcal{V}^l|}$: adjacency matrix for feature aggregation in layer l $(1 \leqslant l \leqslant L)$. $\boldsymbol{A}^l$ defines the inter-layer connectivity between $\mathcal{V}^{l-1}$ and $\mathcal{V}^l$.
- **Aggregate()** function that is used by each vertex to aggregate information from its neighbors.
- **Update()** function including an one-layer multi-layer perceptron (MLP) and an activation function $\sigma()$ that is used to perform feature update.
- $\boldsymbol{W}^l \in \mathbb{R}^{f^{l-1} \times f^l}$: weight matrix of layer l $(1 \leqslant l \leqslant L)$ that is used in update function to perform linear transformation of vertex features.
- $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}| \times f^l}$: input feature matrix.
- $\boldsymbol{h}^l \in \mathbb{R}^{|\mathcal{V}^l| \times f^l}$: the vertex matrix in layer l $(0 \leqslant l \leqslant L)$. Moreover, $\boldsymbol{h}^0 = \boldsymbol{X}$.

GNNs learn to generate low-dimensional vector representation (i.e., node embedding) for a set of target vertices $\mathcal{V}^t$. We illustrate the above process in Fig. 1 with an example of a L-layer GNN model. Starting from layer 1, the GNN model computes the feature vector of each vertex in $\mathcal{V}^1$ by aggregating and updating the feature vectors of its neighbor vertices in $\mathcal{V}^0$; this process is repeated L times until the node embedding of the target vertices $\mathcal{V}^t$ (which is $\mathcal{V}^L$) is derived. The derived node embedding capture the structural information $\boldsymbol{A}$ and vertex features $\boldsymbol{X}$ of the input graph and can be used to facilitate many downstream applications as mentioned in Sect. 1.

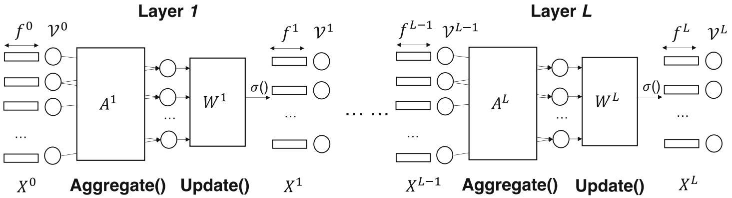

Fig. 1. GNN computation abstraction

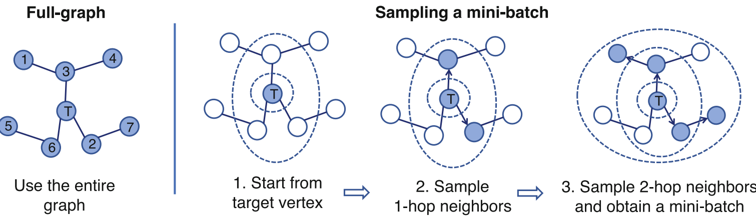

Fig. 2. Full-graph vs. Mini-batch

2.2 Mini-Batch GNN Training

GNNs can be trained in full-graph [15], or in a mini-batch fashion [9,23]. The former approach uses the entire graph to compute the node embeddings of all the vertices; the latter approach first samples a set of vertices and edges and only utilizes the sampled vertices and edges to compute the node embeddings of the target vertices. Mini-batch GNN training demonstrate advantages compared with full-graph training in terms of accuracy, and scalability for large-scale graphs [9,23]; thus, this work focuses on accelerating mini-batch GNN training. We illustrate the difference between the two approaches in Fig. 2, the blue-colored vertices are selected to compute the node embedding of the target vertex (labeled with the letter "T"). Note that there are various mini-batch sampling algorithms [19], Fig. 2 only depicts the Neighbor Sampling algorithm [9] for simplicity. It is also worth noticing that the numbers labeled on the vertices in Fig. 2 are in random order since graph-structured data is non-Euclidean. Since accessing the vertices in random order incurs random memory access, GNN training suffers from high communication overhead. The mini-batch training process consists of five stages [9,23]: sampling, forward propagation, loss calculation, back propagation, and weight update. In the sampling stage, a set of vertices and adjacency matrices are sampled from the input graph topology $\mathcal{G}(\mathcal{V}, \mathcal{E})$. We use $\mathcal{V}^l$ to denote the vertices sampled from $\mathcal{V}$ in layer l. $\boldsymbol{A}^l$ denotes the sampled adjacency matrix, which describes inter-layer connections (edges) between $\mathcal{V}^{l-1}$ and $\mathcal{V}^l$ within the mini-batch. A mini-batch consists of target vertices $\mathcal{V}^t$, sampled vertices for each layer $\{\mathcal{V}^l : 0 \leqslant l \leqslant L-1\}$, and sampled adjacency matrices $\{\boldsymbol{A}^l : 1 \leqslant l \leqslant L-1\}$. In the forward propagation stage, the mini-batch is processed layer by layer as in Fig. 1. The node embeddings in the last layer $\{\boldsymbol{h}_i^L : v_i \in \mathcal{V}^L\}$ are compared with the ground truth for loss calculation. The calculated loss is used for back-

propagation, which performs a similar computation as forward propagation but in a reverse direction. At last, the gradients of $\boldsymbol{W}^l$ in each layer are derived and used to update the weights.

Algorithm 1. Mini-batch GNN Training Algorithm

```
1: for each iteration do
2:     Sampling(G(V, E))                          ▷ Derive mini-batches
3:     for l = 1...L do                           ▷ Forward Propagation
4:         for vertex v ∈ V^l do
5:             a_v^l = Aggregate(h_u^{l-1} : u ∈ N_s(v), u ∈ V^{l-1})
6:             h_v^l = Update(a_i^l, W^l, σ())
7:         end for
8:     end for
9:     CalculateLoss({h_i^L : v_i ∈ V^L})
10:    BackPropagation( )                         ▷ Derive gradient of W^l
11:    WeightUpdate( )
12: end for
```

We show the steps of GNN training in Algorithm 1, $\mathcal{N}_s(v)$ denotes neighbors of v in $\mathcal{V}^{l-1}$ that are specified in $\boldsymbol{A}^l$.

2.3 Related Work

Hardware Acceleration for GNN Training. GraphACT [22] accelerates GNN training on a CPU-FPGA heterogeneous platform by exploiting both task-level parallelism and data parallelism. It adopts a redundancy reduction technique to reduce the number of memory access; however, the technique can only be applied to graphs with binary edge weight. Thus, GraphACT cannot support certain GNN models such as Graph Convolutional Network (GCN) [15] with non-binary edge weight. HP-GNN [17] proposes a general framework that is able to accelerate various GNN models. Given a sampling algorithm, GNN model, and platform metadata, the framework automatically generates a GNN training implementation that runs on a CPU-FPGA heterogeneous platform. Though HP-GNN is able to accelerate various GNN models on a CPU-FPGA platform, it does not support CPU+Multi-FPGA heterogeneous platform which needs to address the high FPGA-to-FPGA communication overhead and tackle the workload imbalance issue.

GNN Training Using Multiple CPUs or GPUs. DistDGL [26] accelerates GNN training on a cloud platform with multiple CPU instances. It shows that GNN training on multiple instances with synchronous stochastic gradient descent (SGD) quickly converges to almost the same accuracy as training on a single machine. In addition, DistDGL proposes to use graph partitioning to reduce the communication overhead among different nodes and achieve high training performance. PaGraph accelerates GNN training on a multi-GPU platform. PaGraph partitions the input graph using a greedy algorithm that

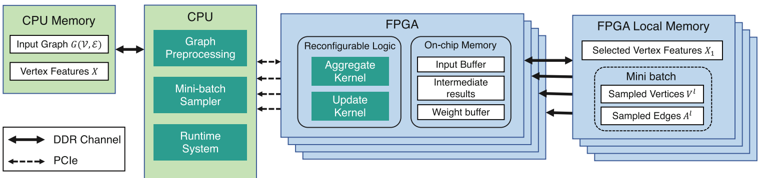

Fig. 3. CPU+Multi-FPGA heterogeneous Platform

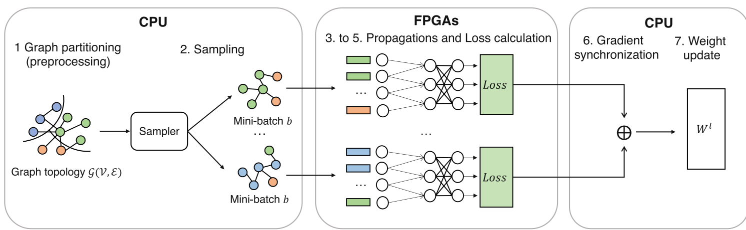

Fig. 4. GNN training on a CPU+Multi-FPGA heterogeneous platform

balances the workload among partitions. In addition, it caches vertex feature of high out-degree vertices since these vertices are expected to be frequently accessed. Utilizing multiple CPUs or GPUs, these works improve GNN training performance compared with a single CPU or GPU. However, as mentioned in Sect. 1, training GNNs using general-purpose processors with fixed data access patterns and complex memory hierarchy suffers from inefficiency; this motivates us to accelerate GNN training on a CPU+Multi-FPGA heterogeneous platform, which is not yet explored by any previous work.

3 GNN Training on CPU+Multi-FPGA Platform

We illustrate a CPU+Multi-FPGA Heterogeneous Platform in Fig. 3. The platform consists of a multi-core CPU connected to the CPU memory via DDR memory channel. The CPU is connected to multiple FPGAs via PCIe. Each FPGA has a local DDR memory.

We depict the workflow of GNN training on a CPU+Multi-FPGA heterogeneous platform in Fig. 4. The training algorithm on a CPU+Multi-FPGA heterogeneous platform is similar to Algorithm 1, but with two additional stages: graph preprocessing and gradient synchronization. We assign the CPU to perform graph preprocessing since the preprocessing is well-supported by existing library[1]. Thus, we store the input graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X})$ in the CPU memory for the CPU to perform graph preprocessing. Note that the preprocessing overhead can be amortized since the graph partitioning is a one-time cost. During the graph processing phase, the input graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X})$ is partitioned and distributed to

[1] https://github.com/KarypisLab/METIS.

each FPGA, we use $\boldsymbol{X_i}$ to indicate the vertex features stored in the ith FPGA local DDR memory. We use different colors to indicate that the vertices are assigned to different FPGAs in Fig. 4. When the graph preprocessing is done, the five stages in Algorithm 1 are performed. We assign the CPU to perform mini-batch sampling because it is flexible to support various sampling algorithms. The sampler samples from each graph partition, and assigns the mini-batches to each FPGA. Note that it is possible to sample vertices from different graph partitions as shown in the mini-batches of Fig. 4; because the edges crossing graph partitions are preserved in a graph partition, so the sampler might sample some vertices in other partitions via the partition-crossing edges. Although accessing vertices in other graph partitions incurs FPGA-to-FPGA communication, preserving the edges crossing different partitions is necessary since removing them would affect the training accuracy. After the mini-batches are produced and distributed, each FPGA performs forward propagation, loss calculation, and back propagation in parallel; we assign FPGAs to perform the GNN operations because the optimized hardware kernels can efficiently deal with the irregular data access patterns in GNNs. Thus, we store the mini-batch topology $\mathcal{V}^l$, $\boldsymbol{A}^l$, and selected vertex features $\boldsymbol{X}_i$ in the FPGA local DDR memory to perform GNN operations. Finally, we assign the CPU to perform gradient synchronization and weight update since it's easier to synchronize using the CPU.

We perform GNN training using synchronous stochastic gradient descent (SGD) [8], which is widely used in related works that accelerate GNN training on a multi-GPU platform. We accelerate the GNN computations but do not alter the training algorithm; thus, the convergence rate and the accuracy are the same as training on a multi-GPU platform using synchronous SGD.

4 Optimizations

4.1 Graph Partitioning and Workload Balancing

Assume there are p FPGAs on the target platform, we partition the input graph into p partitions, and store each partition in one FPGA local DDR memory. During feature aggregation (Algorithm 1), the vertex features of the neighbor vertices are fetched and aggregated. If the vertex required resides in the same graph partition, the vertex feature can be fetched directly from the local DDR memory; otherwise, the vertex feature needs to be fetched from another DDR memory, which incurs high overhead FPGA-to-FPGA communication.

To minimize the FPGA-to-FPGA communication overhead, we utilize METIS [13] algorithm to perform graph partitioning. METIS can minimize cross-partition edge connection and thus reduce FPGA-to-FPGA communication overhead. However, each graph partition consists of a different number of vertices and edges; thus, the workload of training on each graph partition is also different. We develop a task scheduler to balance the workload among FPGAs. Figure 5 illustrates the idea with an example of 4 FPGAs. First, a Mini-batch

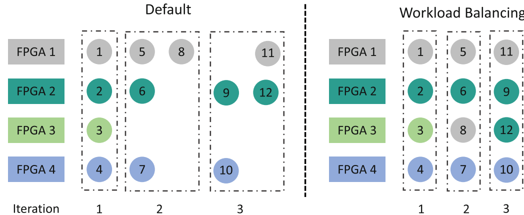

Fig. 5. Workload balancing scheduler

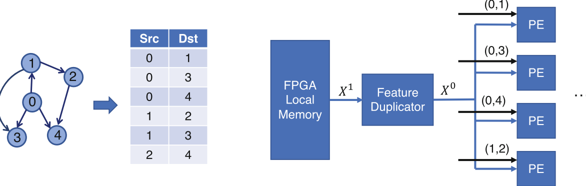

Fig. 6. Data structure

Sampler samples a mini-batch from each graph partition in a round-robin fashion. Each circle in the figure represents a mini-batch, the labeled number indicates the order that each mini-batch is produced, and the color indicates from which graph partition it is sampled. 4 mini-batches is executed in each training iteration, and then a synchronized SGD is performed to update the model weights. In iteration 2, all the mini-batches in partition 3 have been executed. Thus, the sampler continues to sample another mini-batch to produce 4 mini-batches. By default, mini-batch 8 should be computed by FPGA 1 according to the graph partitioning, which causes workload imbalance. Our task scheduler addresses the workload imbalance by assigning the additional mini-batches to idle FPGAs. For example, on the right side of Fig. 5, mini-batch 8 is assigned to FPGA 3. Similarly, in iteration 3, an additional mini-batch is sampled from partition 2 and is then assigned to FPGA 3. Note that this workload balancing technique does not alter the algorithm. As we can see in Fig. 5, the mini-batches being computed in each iteration are the same in both scheduling strategies.

4.2 Optimized GNN Kernels

As mentioned in Sect. 2.2, each GNN layer performs two major steps: feature aggregation and feature update. The aggregation kernel fetches the feature vectors of source vertices, performs an aggregation function which depends on the GNN model, and then writes the result to the destination vertices; the update kernel performs a multi-layer perceptron, which can be implemented using matrix multiplication, to update the feature vectors of the destination vertices. In this subsection, we focus on the optimizations done in the aggregate kernel since it is the bottleneck of GNN training; for the update kernel, we adopt

a systolic-array-based design to perform matrix multiplication of feature matrix $\boldsymbol{h}^l$ and weight matrix $\boldsymbol{W}^l$.

Data Structure. GNN training suffers from poor data reuse, resulting in frequent accesses to the FPGA local DDR memory, which is much slower than accessing on-chip memory like Block RAMs (BRAMs). To exploit data reuse, we store the graph edges in coordinate (COO) format, sorted by the source vertices; this allows our kernels to maximize the opportunity for data reuse. We illustrate the idea in Fig. 6 with a toy example. During GNN training, the aggregation kernel first fetches the feature vector of v_0 from the local DDR memory, a Feature Duplicator will then duplicate the feature vector and store it inside each PE's register. In the meantime, each PE reads an edge to compute. If the source value of the edge matches the feature vector stored inside the PE register (e.g., the first three PE in Fig. 6), then the PE proceeds with its computation of multiplying edge weight with feature vector; if a mismatch occurs (e.g., the 4th PE), the PE stalls and waits for the next feature vector (e.g., feature vector of vertex 1) to compute. Using the proposed data structure and hardware design, the feature vector of each source vertex only needs to be fetched once from the DDR memory, which reduces the communication cost from $O(|\boldsymbol{A}^1|)$ to $O(|\mathcal{V}^0|)$.

Memory Organization and Datapath. Sorting the graph edges by source vertices allows us to exploit data reuse, but also incurs random memory write since the destination vertices are in random order. To mitigate the overhead of random memory write, the aggregate kernel buffers the intermediate results on-chip instead of writing them back to the DDR memory; this allows the intermediate results to be stored in one cycle. After the aggregation is done, the aggregated results are directly transferred to the update kernel. Similarly, after the update is done, the updated results are directly transferred to the aggregate kernel for the computation of next layer. After all L layers are executed, the final results are written back to the local DDR memory sequentially. Utilizing the FPGA on-chip memory to buffer the intermediate results, we reduce the overhead of the random memory access; the datapath design allows the kernels to directly read input data in one cycle since the data do not need to travel through a complex memory hierarchy. In addition, the datapath avoids frequent access to the local DDR memory because it does not need to write back the intermediate results.

5 Experiments

5.1 Experimental Setup

Environments. We run our experiments on a dual-socket server. For the multi-GPU platform, we equip the server with 4 GPUs; and for the CPU-Multi-FPGA heterogeneous platform, we equip the server with 4 FPGAs. The GPUs or FPGAs are connected to the host CPU via PCIe. We list the information of

Table 1. Specifications of the platforms

Platforms	CPU AMD EPYC 7763	GPU Nvidia RTX A5000	FPGA Xilinx Alveo U250
Technology	TSMC 7 nm+	Samsung 8 nm	TSMC 16 nm
Frequency	2.45 GHz	2000 MHz MHz	300 MHz
Peak performance	3.6 TFLOPS	27.8 TFLOPS	0.6 TFLOPS
TDP power	280 W	230 W	225 W
On-chip memory	256 MB L3 cache	6 MB L2 Cache	54 MB
Memory bandwidth	205 GB/s	768 GB/s	77 GB/s

Table 2. Statistics of the datasets and GNN-layer dimensions

Dataset	#Vertices	#Edges	f_0	f_1	f_2
Reddit (RD)	232,965	23,213,838	602	128	41
Amazon (AM)	1,569,960	264,339,468	200	128	107
ogbn-products (PR)	2,449,029	61,859,140	100	128	47

the host CPU, GPUs, and FPGAs in Table 1. Note that the peak performance and memory bandwidth of FPGA is significantly lower than GPU; thus, the speedup of our work highly relies on our optimizations. We develop our hardware kernels using Xilinx Vitis HLS v2021.2, and implement the host program using C++14 with the openCL library. We implement the multi-GPU baseline using Python v3.6, PyTorch v1.11, CUDA v11.3, and PyTorch-Geometric v2.0.3.

Measurements. We use the built-in *time*[2] library to measure the GNN training time on the multi-GPU platform. We build a cycle-accurate simulator to measure the GNN training time on the CPU-multi-FPGA heterogeneous platform. To verify the simulator, we implement the host program and GNN kernels, measure the program execution time on the CPU and post-synthesis execution time on the FPGA using the *time* library, and then tune our simulator according to the data we collected from the actual hardware. We use the Vitis Analyzer [14] to obtain the power consumption of the FPGAs. Vitis Analyzer creates a power trace report, and the power consumption can be calculated using the report. We use Nvidia System Management Interface (SMI) [5] and PowerTop tool [6] to monitor the power consumption of GPUs and CPUs, respectively; these two tools only monitor the power usage instead of providing a power trace report. Thus, we create a script to trace the power consumption to manually obtain the power trace report. Since the sample period of Nvidia SMI is between 1/6 to 1 s, we set our script to read the power consumption from SMI every 0.1 s. In all of our experiments, we measure the data by training 10 epochs and taking the average of the measured value. In our experiments, the observed variation for each epoch is approximately the same (relative standard deviation less than

[2] https://docs.python.org/3/library/time.html.

5%), so measuring the values from 10 epochs is similar as measuring the values from all the epochs that it takes for the model to converge.

GNN Models and Datasets. We run our experiments using two well-known GNN models: GraphSAGE (GSG) [9] and GCN [15]. We use a 2-layer model with a hidden feature size of 128 for all the tasks since this is a widely-used setup [9,24]. We choose three datasets with over 10 million edges for evaluation, namely the Reddit dataset (RD), Amazon dataset (AM) [23], and ogbn-products (PR) [10]. We use the Neighbor Sampler [9] to produce mini-batches; we set the size of target vertices $|\mathcal{V}^t|$ as 1024, the neighbor sampling size of each layer is 25 and 10, and the learning rate is 0.01. Note that under the setup of synchronized SGD, training 4 mini-batches of size 1024 in parallel is equivalent to training a mini-batch of size 4096 on a single GPU or FPGA. Details of the datasets and the GNN-layer dimensions are shown in Table 2.

5.2 Hardware Parameter Selection and Resource Utilization

There are two parameters in our kernel design. We use n and m to denote the parallelism of the aggregate kernel and update kernel, respectively. In particular, n indicates the number of processing elements (PEs) in the aggregate kernel. Figure 6 shows an example of n equals 4. m indicates the number of multiply-and-accumulate (MAC) units in the systolic-array-based kernel design.

Given a GNN model, we aim to find a set of parameters that optimizes the throughput. We first assign an initial value for n and m, evaluate its performance on the three datasets (Sect. 5.1), and observe which kernel is the bottleneck. Then we increase the parallelism of the bottleneck kernel and re-evaluate the performance. We repeatedly increase the parallelism of the bottleneck kernel in each iteration until we saturate the available hardware resources. Both the GCN model and the GraphSAGE model lead to the same set of parameters when the hardware resources are saturated. We show the selected parameters and resource utilization in Table 3.

Table 3. Hardware parameters and resource utilization

Parallelism (n, m)	LUTs	DSPs	URAM	BRAM
(8, 2048)	72%	90%	48%	40%

5.3 Performance Metrics

- Epoch time: the time it takes to train one epoch (seconds).
- Throughput: we define the training throughput as the Number of Vertices Traversed Per Second (NVTPS).
- Bandwidth efficiency: throughput divided by available memory bandwidth of the target platform (NVTPS/(GB/s)). Since the bandwidth varies on different platforms, normalizing the throughput with the available bandwidth provides a clear indication of the effectiveness/efficiency of the accelerator.

Table 4. Comparison with multi-GPU platform

			RD	AM	PR	Geo. Mean
GCN [15]	Epoch time	GPU	1.21	4.04	4.61	–
		This work	0.57	1.05	2.81	–
	Throughput	GPU	25.3 M	27.6 M	106 M	42.0 M (1×)
		This work	53.8 M	107 M	175 M	100 M (2.38×)
	BW efficiency	GPU	7.71 K	8.42 K	32.5 K	12.8 K (1×)
		This work	105 K	208 K	340 K	195 K (15.2×)
	Energy efficiency	GPU	0.47	1.58	1.80	1.10 (1×)
		This work	0.12	0.22	0.59	0.25 (4.40×)
GSG [9]	Epoch time	GPU	1.25	4.16	4.89	–
		This work	0.71	1.78	4.27	–
	Throughput	GPU	24.4 M	26.8 M	100 M	40.4 M (1×)
		This work	42.9 M	62.7 M	115 M	67.6 M (1.67×)
	BW efficiency	GPU	7.46 K	8.17 K	30.6 K	12.3 K (1×)
		This work	83.6 K	122 K	224 K	132 K (10.7×)
	Energy efficiency	GPU	0.49	1.63	1.91	1.15 (1×)
		This work	0.15	0.38	0.90	0.37 (3.10×)

– Energy efficiency: the energy consumption of training one epoch on the target platform (kJ/epoch).

5.4 Comparison with Multi-GPU Platform

Performance. We compare the performance of our design on a CPU+Multi-FPGA heterogeneous platform, with a state-of-the-art GNN training implementation using PyTorch-Geometric on a multi-GPU platform. Both the multi-GPU baseline and our work adopts the METIS algorithm for graph preprocessing. In our work, we overlap the sampling stage and GNN operations in each training iteration since they are performed on CPU and FPGAs, respectively. We use the performance metrics defined in Sect. 5.3 to compare with the multi-GPU baseline. We list the results in Table 4. As noted in Sect. 5.1, we obtain the experimental results by training 10 epochs and then average the measured values. The measured values from each epoch are very close to each other: the maximum relative standard deviation in our experiments is 3.3%. We use *GPU* to indicate the multi-GPU baseline, and use *This work* to indicate our work which runs on the CPU+Multi-FPGA heterogeneous platform. We achieve 2.38× and 1.67× speedup on the GCN model and GraphSAGE model, respectively; this is because (1) our task scheduler balances the workload on each FPGA which reduces the parallel execution time; and (2) our optimized GNN kernels effectively reduce the memory access overhead.

Note that GPUs have much higher peak performance and memory bandwidth than FPGAs; thus, to illustrate the effectiveness of our optimizations, we further compare the bandwidth efficiency on both platforms which normalized

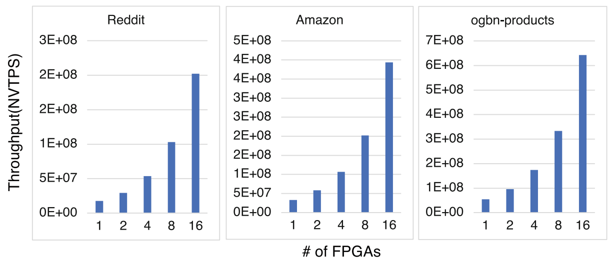

Fig. 7. Throughput scales linearly w.r.t. number of FPGAs on the target platform

the throughput with the available bandwidth on the platform. We achieve up to 24.7× bandwidth efficiency than the multi-GPU baseline; in other words, our design is able to achieve up to 24.7× throughput given the same memory bandwidth. While the workload balancing technique can also be applied to the multi-GPU platform, the GNN kernel optimizations are specific for FPGAs. If we apply the same data structure to the multi-GPU baseline, we are able to exploit some data reuse with the GPU cache since the edges are sorted by the source vertices; however, we can not mitigate the random memory write overhead since we have no control over the datapath and on-chip memory on a GPU platform. On an FPGA platform, we overcome this issue by utilizing the abundant on-chip memory to buffer the intermediate results; we also design a datapath to avoid any unnecessary write to the local DDR memory during GNN training (Sect. 4.2). Thus, even if we speed up the memory read phase on the multi-GPU platform, the memory write bottlenecks the performance.

In addition, unlike our kernels which can access the data in one cycle (3.3 ns), GPUs require multiple cycles to access the data in multi-level caches. Taking Nvidia RTX 3090 as an example, the L2 cache latency is over 130 ns [4]. Note that we use Nvidia RTX A5000 for our experiments, which uses the same GPU architecture (GA102) as Nvidia RTX 3090, so we expect similar cache latency on both GPUs. Finally, our work achieves up to 7.18× energy efficiency than the multi-GPU baseline. This is because our dedicated hardware designs can efficiently perform GNN training, while GPUs launch massive amount of CUDA cores with low utilization.

Convergence. As mentioned in Sect. 3, our work does not alter the original training algorithm; thus, the convergence rate of our work is the same as the serial training algorithm.

5.5 Scalability

We evaluate the scalability of our work using the three datasets on a two-layer GCN model. As shown in Fig. 7, our work achieves a scalable speedup as we increase the number of FPGAs. We do not consider cases with more than 16 FPGAs since it exceeds the number of PCIe channels on our target platform.

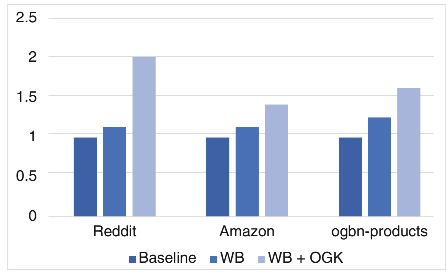

Fig. 8. Throughput improvement due to optimizations

5.6 Impact of Optimizations

We evaluate the two optimizations of workload balancing (WB) and optimized GNN kernels (OGK) described in Sect. 4 on a two-layer GCN model. We first measure the throughput of the baseline implementation with no optimizations, and then incrementally apply the two optimizations. The throughput in Fig. 8 is normalized with the baseline design. Both optimizations increase the GNN training throughput and can deliver up to 2× improvement in total.

6 Conclusion

In this work, we accelerated GNN training using a CPU+Multi-FPGA heterogeneous platform. We developed several techniques to efficiently accelerate GNN training on our target platform. Using much less compute power and memory bandwidth than GPUs, our work achieved up to 2.38× speedup and 4.40× energy efficiency compared with the multi-GPU baseline due to the 24.7× bandwidth efficiency. In the future, we plan to extend our work to a general framework that can automatically map GNN training on any given CPU+Multi-FPGA platform.

Acknowledgement. This work has been supported by the U.S. National Science Foundation under grant number OAC-2209563.

References

1. Amazon ec2 f1. https://aws.amazon.com/tw/ec2/instance-types/f1/. Accessed 23 June 2022
2. Azure np-series. https://docs.microsoft.com/en-us/azure/virtual-machines/np-series. Accessed 23 June 2022
3. Intel devcloud. https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html. Accessed 23 June 2022
4. Measuring GPU memory latency. https://chipsandcheese.com/2021/04/16/measuring-gpu-memory-latency/. Accessed 20 June 2022
5. Nvidia system management interface. https://developer.nvidia.com/nvidia-system-management-interface. Accessed 21 June 2022

6. PowerTOP. https://github.com/fenrus75/powertop. Accessed 21 June 2022
7. Cai, Z., Yan, X., Wu, Y., Ma, K., Cheng, J., Yu, F.: DGCL: an efficient communication library for distributed GNN training. In: 16th European Conference on Computer Systems (2021)
8. Chen, J., Monga, R., Bengio, S., Jozefowicz, R.: Revisiting distributed synchronous SGD. In: International Conference on Learning Representations Workshop (2016)
9. Hamilton, W.L., Ying, R., Leskovec, J.: Inductive representation learning on large graphs. In: 31st Neural Information Processing Systems (2017)
10. Hu, W., et al.: Open graph benchmark: datasets for machine learning on graphs. arXiv preprint arXiv:2005.00687 (2020)
11. Huang, K., Zhai, J., Zheng, Z., Yi, Y., Shen, X.: Understanding and bridging the gaps in current GNN performance optimizations. In: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2021 (2021)
12. Jiang, W., Luo, J.: Graph neural network for traffic forecasting: a survey. arXiv preprint arXiv:2101.11174 (2021)
13. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**, 359–392 (1998)
14. Kathail, V.: Xilinx vitis unified software platform. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (2020)
15. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations (2017)
16. Lin, Y.C., Zhang, B., Prasanna, V.: GCN inference acceleration using high-level synthesis. In: IEEE High Performance Extreme Computing Conference (2021)
17. Lin, Y.C., Zhang, B., Prasanna, V.: HP-GNN: generating high throughput GNN training implementation on CPU-FPGA heterogeneous platform. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (2022)
18. Lin, Z., Li, C., Miao, Y., Liu, Y., Xu, Y.: PaGraph: scaling GNN training on large graphs via computation-aware caching. In: ACM Cloud Computing (2020)
19. Liu, X., Yan, M., Deng, L., Li, G., Ye, X., Fan, D.: Sampling methods for efficient training of graph convolutional networks: a survey. IEEE/CAA J. Autom. Sinica **9**, 205–234 (2022)
20. Yan, M., et al.: HYGCN: a GCN accelerator with hybrid architecture. In: International Symposium on High Performance Computer Architecture (HPCA) (2020)
21. Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W.L., Leskovec, J.: Graph convolutional neural networks for web-scale recommender systems. In: 24th ACM SIGKDD Knowledge Discovery & Data Mining (2018)
22. Zeng, H., Prasanna, V.: GraphACT: accelerating GCN training on CPU-FPGA heterogeneous platforms. In: ACM/SIGDA Field-Programmable Gate Arrays (2020)
23. Zeng, H., Zhou, H., Srivastava, A., Kannan, R., Prasanna, V.: GraphSAINT: graph sampling based inductive learning method. In: International Conference on Learning Representations (2020)
24. Zhang, B., Kannan, R., Prasanna, V.: BoostGCN: a framework for optimizing GCN inference on FPGA. In: 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE (2021)
25. Zhang, B., Zeng, H., Prasanna, V.: Hardware acceleration of large scale GCN inference. In: 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP). IEEE (2020)
26. Zheng, D., et al.: DistDGL: distributed graph neural network training for billion-scale graphs. CoRR (2020)