Hardware Acceleration of Large Scale GCN Inference

Bingyi Zhang, Hanqing Zeng, Viktor Prasanna
University of Southern California, Los Angeles, California
Email: {bingyizh, zengh, prasanna}@usc.edu

***Abstract*—Graph Convolutional Networks (GCNs) have become state-of-the-art deep learning models for representation learning on graphs. Hardware acceleration of GCN inference is challenging due to: 1) massive size of the input graph, 2) heterogeneous workload of the GCN inference that consists of sparse and dense matrix operations, and 3) irregular information propagation along the edges during the computation. To address the above challenges, we propose the algorithm-architecture co-optimization to accelerate large-scale GCN inference on FPGA. We first perform data partitioning to fit each partition in the limited on-chip memory. Then, we use a two-phase preprocessing algorithm consisting of *sparsification* and *node reordering*. The first phase (*sparsification*) eliminates edge connections of high-degree nodes by merging common neighbor nodes. The second phase (*re-ordering*) effectively groups adjacent nodes to improve on-chip data reuse. Incorporating the above algorithmic optimizations, we propose a generic FPGA architecture to pipeline the two major computational kernels in GCN: aggregation and transformation. The flexible data path and task scheduling strategy of our design support various GCN models and lead to high throughput inference. We evaluate our design on state-of-the-art FPGA platform using three large scale datasets: Flickr, Reddit, Yelp. Compared with the state-of-the-art multi-core and GPU baselines, our design improves the throughput by up to 30× and 2× respectively.**

I. Introduction

Graph Convolutional Network (GCN) is an emerging type of neural network model, capable of learning features from unstructured graphs [1]. For the task of inference, the input is a node-attributed graph. For each node, the output is an embedding of the node, which is a low-dimensional vector representation of this node. Node embedding can be used for various downstream tasks. For example, in citation network [2], GCN can learn from the labeled nodes and infer the categories of unlabeled ones. In biochemical analysis [3], GCN can learn to identify the properties of unknown proteins based on the existing proteins. In social networks mining [4], GCN can learn from the archived posts to classify the new posts.

With the deployment of GCNs in many real-life applications, hardware acceleration of GCN inference is needed. For example, in the large scale recommendation system used by Pinterest [5], with the continuously updated attributes of users and posts, Pinterest needs to quickly generate embeddings for millions of users to enable real-time recommendations. In e-commerce system used by Alibaba [6], GCN is required to perform fast recommendation for hundreds of millions of users. In traffic network, the GCN is used to predict the future traffic flows based on the historical traffic flows [7] in real time.

Essentially, GCN consists of a stack of graph convolution layers. Each layer performs two major steps [2]: (1) *Aggregation*: each node aggregates information from its neighbors and itself; (2) *Transformation*: the aggregated feature vector is transformed into another feature space.

While the computational steps of GCN inference are fairly straightforward, accelerating the GCN inference is challenging due to (1) *Massive input graph size*: For GCN inference, the input is the full graph which can be very large (e.g. Amazon dataset [8] has millions of nodes and billions of edges). (2) *Heterogeneous workload*: GCN inference consists of sparse and dense matrix operations *feature aggregation* and *weight transformation* which involve intensive tensor operations. (3) *Irregular data access*: Unlike traditional convolutional neural network (CNN) which has regular and predictable memory access, feature aggregation in GCN leads to irregular data access and thus limited on-chip data reuse. This can lead to large overhead of external memory access, and thus inference performance degradation.

To address these challenges, we propose the algorithm-architecture co-optimization to accelerate large-scale GCN inference on FPGA. First, we perform data partitioning to satisfy the on-chip storage limitation. Then, we develop a two-phase pre-processing algorithm to reduce computational complexity and increase the data locality. In hardware design, we map the key computational kernels on FPGA and their data communication is through the on-chip memory for pipelined execution. The data path supports various GCN models and different computational orders. Our main contributions are as follows:

- We propose a data partitioning scheme to partition the input data; Each data partition can fit in FPGA on-chip memory. We also develop the corresponding scheduling strategy based on this data partition scheme to facilitate efficient GCN inference.
- We propose a two-phase graph pre-processing algorithm to reduce the external memory traffic:
  - **Graph sparsification**: The sparsification phase reduces the redundant edge connections by merging common neighbors. This step reduces the number of external memory accesses. It also facilitates the subsequent re-ordering phase by reducing the edge connections of high-degree nodes.

2160-052X/20/$31.00 ©2020 IEEE
DOI 10.1109/ASAP49362.2020.00019

  - **Node re-ordering**: Node re-ordering phase groups the adjacent nodes by re-assigning the node indices, which increases the data locality and significantly reduces the number of external memory accesses.
- We develop a hardware architecture which can efficiently execute GCN inference:
  - **Pipeline**: We build hardware modules for the key computational kernels: feature aggregation and feature transformation. Their data communication is through the on-chip memory of FPGA which enables pipelined execution.
  - **Scheduling**: Based on our hardware pipeline, we develop a scheduling strategy which efficiently executes the GCN inference. It also supports the execution of various GCN models.
  - **Flexibility**: We identify the two computation orders in GCN inference. Based on that, we build the flexible data path which supports the two computational orders.
- We provide the mathematical analysis of data communication cost, which accurately models the impact of data partitioning and two-phase pre-processing algorithm on the memory traffic of our proposed hardware architecture.
- We evaluate our hardware architecture on three large-scale datasets. The experimental results show that compared with multi-core CPU and high-end GPU baselines, our hardware architecture achieves throughput up to 30× and 2× respectively. The two-phase pre-processing algorithm leads to nearly 30% performance improvement.

II. Background

TABLE I
THE STATE-OF-THE-ART GCN MODELS

Approach	Forward Rule
Vanilla-GCN [2]	$\boldsymbol{X}^{l+1} = \phi(\hat{\boldsymbol{D}}^{-\frac{1}{2}}\hat{\boldsymbol{A}}\hat{\boldsymbol{D}}^{-\frac{1}{2}}\boldsymbol{X}^{l}\boldsymbol{W}^{l})$ †
GraphSAGE [4]	$\boldsymbol{X}^{l+1} = \phi(\boldsymbol{X}^{l}\boldsymbol{W}^{l}_{\text{self}}\vert\boldsymbol{D}^{-1}\tilde{\boldsymbol{A}}_{l}\boldsymbol{X}^{l}\boldsymbol{W}^{l}_{\text{neighbor}})$ ‡
GraphSAINT [8], [9]	$\boldsymbol{X}^{l+1} = \phi(\boldsymbol{X}^{l}W^{l}_{\text{self}}\vert\boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{X}^{l}\boldsymbol{W}^{l}_{\text{neighbor}})$
FastGCN [10]	$\boldsymbol{X}^{l+1} = \phi(\hat{\boldsymbol{D}}^{-\frac{1}{2}}\hat{\boldsymbol{A}}\hat{\boldsymbol{D}}^{-\frac{1}{2}}\boldsymbol{X}^{l}\boldsymbol{W}^{l})$ †
Cluster-GCN [11]	$\boldsymbol{X}^{l+1} = \phi((\bar{\boldsymbol{A}} + \lambda \cdot diag(\bar{\boldsymbol{A}}))\boldsymbol{X}^{l}\boldsymbol{W}^{l})$ *

† $\phi(\cdot)$ is the activation function. $\boldsymbol{I}_N + \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}} \rightarrow \hat{\boldsymbol{D}}^{-\frac{1}{2}}\hat{\boldsymbol{A}}\hat{\boldsymbol{D}}^{-\frac{1}{2}}$
‡ $\tilde{\boldsymbol{A}}_l$ is the adjacency matrix after edge sampling in GraphSAGE [4].
* $\bar{\boldsymbol{A}} = (\boldsymbol{D} + \boldsymbol{I})^{-1}(\boldsymbol{A} + \boldsymbol{I})$

A. GCN Model

Graph Convolutional Network (GCN) is one of the most popular types of graph neural networks [1] for data in non-Euclidean space. The input to the GCN is a node-attributed graph $\mathcal{G}\left(\mathcal{V}, \mathcal{E}, \boldsymbol{X}\right)$, where $\mathcal{V}$ and $\mathcal{E}$ denote the set of nodes and the set of edges respectively. Each node $v \in \mathcal{V}$ is attributed by a feature vector of length f, and we use $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}| \times f}$ to denote the feature matrix where each row is the feature vector of a node. Suppose the GCN has L graph convolution layers. Then $\boldsymbol{X}$ is the input to the first layer, and we use $\boldsymbol{X}^l \in \mathbb{R}^{|\mathcal{V}| \times f_l}$ to denote the input to the l^{th} layer (i.e., $\boldsymbol{X}^1 = \boldsymbol{X}$). The output of L^{th} layer is $\boldsymbol{X}^{L+1} \in \mathbb{R}^{|\mathcal{V}| \times f_{L+1}}$ where each row is the embedding of a node. Here, we use $\boldsymbol{A}$ to represent the binary adjacency matrix of $\mathcal{G}$. So $\boldsymbol{A}_{uv} = 1$ if edge $(u, v) \in \mathcal{E}$, and $\boldsymbol{A}_{uv} = 0$ otherwise. $\boldsymbol{D}$ is the diagonal degree matrix of $\mathcal{G}$. So $\boldsymbol{D}_{ii} = \sum_{j=1}^{|\mathcal{V}|} \boldsymbol{A}_{ij}$. Each layer l is parameterized by a weight matrix $\boldsymbol{W}^l \in \mathbb{R}^{f_l \times f_{l+1}}$ to transform the aggregated feature vectors. The layer operations of various GCN models are shown in Table I. Various techniques may use slightly different formulas to normalize the adjacency matrix. There are two main computational kernels in GCN models:

- *Feature aggregation (FA)*: $\boldsymbol{AX}$. Here, we use $\boldsymbol{A}$ to denote the adjacency matrix or the normalized adjacency matrix. A row of adjacency matrix can be viewed as the adjacency list of a node which contains information of edge connections. In feature aggregation, each node aggregates the features from its neighbors.
- *Feature transformation (FT)*: $\boldsymbol{XW}$. In feature transformation stage, each feature vector in $\boldsymbol{X}$ is multiplied by the weight matrix $\boldsymbol{W}$. Papers [4], [9] use the self-weight $\boldsymbol{W}^l_{\text{self}} \in \mathbb{R}^{f_l \times \frac{f_{l+1}}{2}}$ to transform the original features and use the neighbor-weight $\boldsymbol{W}^l_{\text{neighbor}} \in \mathbb{R}^{f_l \times \frac{f_{l+1}}{2}}$ to transform the aggregated features. Both of them belong to feature transformation.

B. GCN Acceleration

While there exist many CNN/MLP accelerators [12], [13], few works have been proposed to accelerate GCN inference. Zeng [9] develops a scalable and efficient framework to accelerate GCN training on a multi-core CPU platform. GraphACT [14] accelerates GCN training on CPU-FPGA heterogeneous platform. Our work is different from GraphACT in that 1) GraphACT focuses on accelerating one specific GCN training algorithm–GraphSAINT [15], while our work proposes a generic method to accelerate GCN inference. 2) GraphACT targets at GCN training phase which uses mini-batch method. In each training epoch, a mini-batch subgraph is sampled from the full graph which can fit FPGA on-chip memory. Our work targets at GCN inference phase which is performed in full–batch manner. The input for GCN inference is the full graph, which can not fit FPGA on-chip memory. 3) The mini-batch training method used by GraphACT samples subgraph in each training epoch. The sampled subgraph has significant lower average degree than the full graph used by GCN inference. In full graph, the average degree can be significant higher than the small subgraph and there also exists high-degree nodes, which leads large overhead of communication cost for feature aggregation. Our work aims to address these challenges.

III. Problem Definition

Scope The scope of this paper is to accelerate the GCN models where the feature aggregation $\boldsymbol{AX}$ and feature transformation $\boldsymbol{XW}$ are the key computational kernels as indicated in Table I.

This paper deals with the large, node-attributed, static graph $\mathcal{G}\left(\mathcal{V}, \mathcal{E}, \boldsymbol{X}\right)$. Specifically, the size of the graph $\mathcal{G}$ can be very

TABLE II
STATISTICS OF THE DATASET

Dataset	Number of nodes $\lvert\mathcal{V}\rvert$	Feature length f	Number of edges $\lvert\mathcal{E}\rvert$	Average degree $\bar{d}$
Flickr	89250	500	449878	10.1
Reddit	232965	602	11606919	99.6
Yelp	716847	300	27907940	19.5

large as shown in Table II which exceeds the size of FPGA on-chip memory. For example, Yelp has $700K$ nodes and $27M$ edges. Static graph $\mathcal{G}$ has the property that the features' values in $\boldsymbol{X}$ change over time, while the graph topology (i.e., $\mathcal{V}$, $\mathcal{E}$) is relatively fixed. We assume $\mathcal{G}$ is static, because for many real-life graphs, their graph structures are fixed or relatively stable compared with node's attributes. For example, in social networks, the user activities/posts (encoded as $\boldsymbol{X}$) are updated frequently, while the user relationships (represented by $\mathcal{E}$) are much more stable. In a road network, the traffic condition of a road changes much more dramatically and frequently than the connectivity of roads [16]. As a result, the time for pre-processing the graph is amortized and we do not include the pre-processing time in the execution time.

Metric In graph analytics, the Million Traversed Edges Per Second (MTEPS) is the measure of the throughput [17]. Similarly, we define the throughput of GCN inference system as **Million Embedded Nodes Per Second** (**MENPS**). MENPS denotes the number of nodes which are embedded per second.

Objective Given a large, node-attributed graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X})$, a trained GCN model and a target FPGA device, we aim at increasing the throughput of GCN inference.

Notations We use $\boldsymbol{X}(u)$ to denote the u^{th} row of $\boldsymbol{X}$ and $\boldsymbol{X}(u : v)$ denotes the sub-matrix from u^{th} row to v^{th} row. Similarly, $\boldsymbol{X}(:, v)$ denotes the v^{th} column and $\boldsymbol{X}(:, u : v)$ denotes the submatrix from u^{th} column to v^{th} column. To simplify notation in the rest of the paper, we let $n = |\mathcal{V}|$.

Data sparsity The sparsity of adjacency matrix denotes the fraction of non-zero elements. The adjacency matrix $\boldsymbol{A}$ in GCN can be highly sparse. For example, in Table II, adjacency matrix of PPI, Flickr, Reddit have the sparsity of 0.5%, 0.2%, 0.2% respectively. The density of the weight matrix $\boldsymbol{W}$ approximates 100% as indicated in [18]. For the feature matrix $\boldsymbol{X}$, we treat it as dense matrix due to 1) the input feature matrix $\boldsymbol{X}^1$ is normally dense. 2) the density of $\boldsymbol{X}^l$ $(l > 1)$ is unpredictable depending on the dataset [18] which ranges from 1%-60%. 3) According to [1], many GCN models use LeakyReLU activation function [19] which can make $\boldsymbol{X}$ dense. Since the density of $\boldsymbol{X}$ highly depends on GCN model and the input graph, the exploration of the sparsity in $\boldsymbol{X}$ is not the focus of this paper. We leave it for the future work.

Approach The key operation to compute a GCN layer is $\boldsymbol{A}\boldsymbol{X}^l\boldsymbol{W}^l$, which can be further broken down into $\boldsymbol{C}^l = \boldsymbol{A}\boldsymbol{X}^l$ and $\boldsymbol{Z}^l = \boldsymbol{C}^l\boldsymbol{W}^l$. Since the graph is large and sparse, computation of $\boldsymbol{C}^l$ incurs large volume of random accesses into external memory. On the other hand, while the dense matrices $\boldsymbol{C}^l$ and $\boldsymbol{W}^l$ lead to high on-chip data reuse and simple dataflow, the workload to compute $\boldsymbol{Z}^l$ is very high. Therefore, the problem of GCN inference is both computationally and communicationally intensive.

The key to alleviate the burden of data transfer is to improve data locality and thus on-chip data reuse. We achieve this by pre-processing the graph based on its topology $\mathcal{G}(\mathcal{V}, \mathcal{E})$ (Section IV). The key to realize the fast execution of intensive computation workload is to design a pipelined hardware architecture by exploiting the parallelism underlying GCN layer (Section V).

IV. ALGORITHM ANALYSIS AND OPTIMIZATION

A. Partition-based Inference

The computational kernels–$\boldsymbol{AX}$ and $\boldsymbol{XW}$–need to be mapped on FPGA. However, for real-world datasets, the dimension of $\boldsymbol{A}$ and $\boldsymbol{X}$ can be extremely large. As a result, the $\boldsymbol{A}$ and $\boldsymbol{X}$ can not fit on-chip. Here, we use a data partitioning scheme for GCN inference. We partition the graph by dividing the $\boldsymbol{A}$ along the two dimensions. Here, we denote the submatrix of $\boldsymbol{A}$ as $\boldsymbol{A}_{ij}$, which is assigned the block of adjacency matrix $\boldsymbol{A}\left(i \times \frac{n}{k} : (i+1) \times \frac{n}{k} - 1, j \times \frac{n}{k} : (j+1) \times \frac{n}{k} - 1\right)$. Each of the $\frac{n}{k} \times \frac{n}{k}$ sub-matrices has the size of $k \times k$. Additionally, we partition $\boldsymbol{X}^l, \boldsymbol{C}^l, \boldsymbol{Z}^l$ and $\boldsymbol{X}^{l+1}$ into $\frac{n}{k}$ equal-size parts along the row dimension. For example, $\boldsymbol{X}_i^l = \boldsymbol{X}^l\left(i \times \frac{n}{k} : (i+1) \times \frac{n}{k} - 1\right)$, for $0 \leqslant i < \frac{n}{k}$. In the partition-based inference, calculating the final output $\boldsymbol{X}^{l+1}$ is by sequentially calculating the block $\{\boldsymbol{X}_0^{l+1}, \boldsymbol{X}_1^{l+1}, \boldsymbol{X}_2^{l+1}, ...\}$. Calculating $\boldsymbol{X}_i^{l+1}$ requires the following consecutive steps:

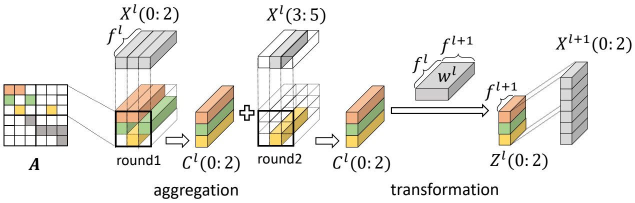


Fig. 1. Partition-based inference.

$$\begin{aligned} &\textcircled{1}\ \boldsymbol{C}_i^l = \sum_{j=0}^{\frac{n}{k}-1} \boldsymbol{A}_{ij}\boldsymbol{X}_j^l \\ &\textcircled{2}\ \boldsymbol{Z}_i^l = \boldsymbol{C}_i^l\boldsymbol{W}^l \quad \textcircled{3}\ \boldsymbol{X}_i^{l+1} = \sigma\left(\boldsymbol{Z}_i^l\right) \end{aligned} \qquad (1)$$

Figure 1 visualizes the computation of equation (1). The adjacency matrix $\boldsymbol{A}$ is partitioned into 4 sub-matrices (i.e., $n = 6$, $k = 3$). Non-zero elements of $\boldsymbol{A}$ are highlighted in color. The aggregation step generates $\boldsymbol{C}_0^l = \boldsymbol{C}^l(0:2)$ after two rounds. Then the transformation step multiplies $\boldsymbol{C}^l(0:2)$ by $\boldsymbol{W}^l$ to generate $\boldsymbol{Z}_0^l = \boldsymbol{Z}^l(0:2)$. Note that the row partitioning of $\boldsymbol{A}$ controls the size of $\boldsymbol{C}_i^l$ and $\boldsymbol{Z}_i^l$, so that the hardware design can satisfy the constraint of on-chip memory. The row partitioning breaks down the aggregation of

$\boldsymbol{X}^l$ into blocks, so that we can pipeline the loading of $\boldsymbol{X}_i^l$ and computation of $\boldsymbol{X}_{i-1}^l$. Details of the hardware design under such 2D partitioning is shown in Section V.

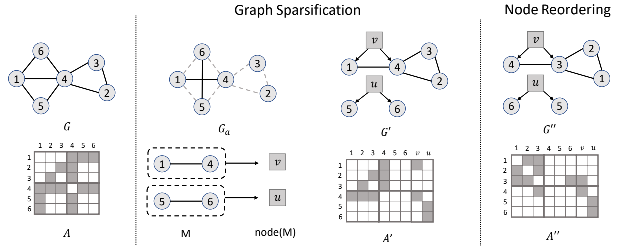


Fig. 2. An example of two-phase pre-processing. The adjacency matrix is partitioned into submatrix of size 3×3 or 3×2. Therefore, the original memory traffic is $2+3+3+3 = 11$. The memory traffic after graph sparsification is $2+1+1+3+2 = 9$. The memmory traffic after node rordering is $3+1+1+1+2 = 8$.

B. Graph Sparsification

In aggregation stage, the memory traffic is related to the number of edges. Intuitively, a graph with more edges requires more memory accesses. Here, we perform graph sparsification by eliminating the "redundant" edges in the graph, without changing the aggregation result, $\boldsymbol{C}^l$. Here, we use the redundancy reduction [14] to eliminate this redundancy. A toy example is indicated in Figure 2. For input graph $\mathcal{G}$, we enumerate the neighbor pairs of each node. For example, node 1 has neighbors $4, 5, 6$. Enumerating all the neighborhood pairs of node 1 gives $(4, 5)$, $(4, 6)$ and $(5, 6)$. Note that $(5, 6)$ is a common pair of neighbors shared by nodes 1 and 4. We identify two common pairs $(1, 4)$, $(5, 6)$ in $\mathcal{G}$. Replace the common pairs with new nodes u, v. Attach u, v with feature vectors $\frac{1}{2}\left(\boldsymbol{X}(5)+\boldsymbol{X}(6)\right)$ and $\frac{1}{2}\left(\boldsymbol{X}(1)+\boldsymbol{X}(4)\right)$. Then, incorporating the new nodes, we delete the redundant edges and construct reduced graph $\mathcal{G}'$. We can do this process for several iterations to further reduce the redundancy.

Compared with its original usage in mini-batch GCN training [14], redundancy reduction brings more benefits to GCN inference: (1) Full graph has more redundancy than the mini-batch subgraph, so that it can reduce considerable number of 'redundant' edges. (2) Sparsification can significantly reduce the edge connections of high degree nodes. This facilitates the node reordering step to be discussed next.

C. Node Reordering

GCN inference has intensive memory traffic due to the randomized data layout, resulting in limited data reuse. Suppose we can increase the data reuse in each submatrix $\boldsymbol{A}_{ij}$, the total external memory access can be reduced. To do this, we change the data layout of the adjacency matrix. Originally, indices of nodes $v_i \in \mathcal{V}$ are assigned randomly. Since in aggregation stage, nodes need to aggregate information from the neighbors, we consider changing the node order by grouping the adjacent nodes together. Using Figure 2 as the example, in $\mathcal{G}'$, due to the randomized node order, nodes $2, 3, 4$, which form a complete subgraph, fall into different partitions of $\boldsymbol{A}$. This results in extra memory accesses when processing $\boldsymbol{A}_{00}$ and $\boldsymbol{A}_{10}$. The cost of processing $\boldsymbol{A}_{00}$ and $\boldsymbol{A}_{10}$ does not decrease after the sparsification phase. After reordering, the nodes $2, 3, 4$ in $\mathcal{G}'$ now become nodes $1, 2, 3$ in $\mathcal{G}''$. Now, the feature propagation and aggregation among the three nodes require processing of a single submatrix $\boldsymbol{A}_{00}$. For this purpose, we use the bandwidth reduction (BR) algorithm proposed in [20] which is a well-known technique to reduce the bandwidth of the matrix by node permutation.

The node reordering algorithm has two properties desirable for GCN inference. Firstly, it increases the data locality by grouping the adjacent nodes together. Thus, data on-chip have higher reuse and accesses to external memroy are reduced. Secondly, it works well with the sparsification phase. We observe that reducing the number of high-degree nodes helps improve the reordering quality. Thus, sparsification can also be considered as a pre-processing step of reordering.

D. Optimization Workflow

We integrate the above two-phase optimizations as a complete pre-processing algorithm. We first run graph sparsification for S iterations. Then, we reorder nodes to increase the data locality. An example is shown in Figure 2. Total number of memory accesses reduces from 11 to 9, and further to 8, after the two-phase preprocessing.

Complexity analysis As explained in Section III, the topology of the static graph is relatively stable compared with the node attributes, the one-shot execution of our pre-processing can benefit many runs of the subsequent GCN inference. For example, in the graph of traffic prediction, the node attributes change over time, but the graph structure is relatively stable. Thus, it is not a critical issue to accelerate pre-processing considering that its cost can be amortized. Nevertheless, to support large-scale graphs, we still require the pre-processing to be computationally tractable. In the following, we analyze the pre-processing complexity. Graph sparsification using the redundant reduction algorithm in [14] has the complexity of $\mathcal{O}\left(S \cdot n \cdot \bar{d}^2\right)$ where $\bar{d}$ is the average degree of the graph. Node reordering using Reversed Cuthill–McKee algorithm [20] has complexity of $\mathcal{O}\left(n \log(n)\right)$. In summary, the two-phase pre-processing can be efficiently and effectively performed.

V. Hardware Architecture

After the algorithmic optimization, we perform data partitioning on $\mathcal{G}$ and $\boldsymbol{A}$. Selecting the appropriate partition size, the on-chip memory can store $\boldsymbol{A}_{ij}$, $\boldsymbol{X}_j^l$ and $\boldsymbol{C}_i^l$ to minimize external memory accesses. In this section, we present an integrated FPGA architecture for GCN inference. Figure 3 shows the overall hardware architecture. The Aggregation, Activation and Transformation modules form a computation pipeline. The Aggregation and Transformation modules consume the majority of the on-chip resources. When the Aggregation module generates $\boldsymbol{C}_i^l$, the results $\boldsymbol{C}_i^l$ can be directly forwarded to the Transformation or Activation modules depending on the

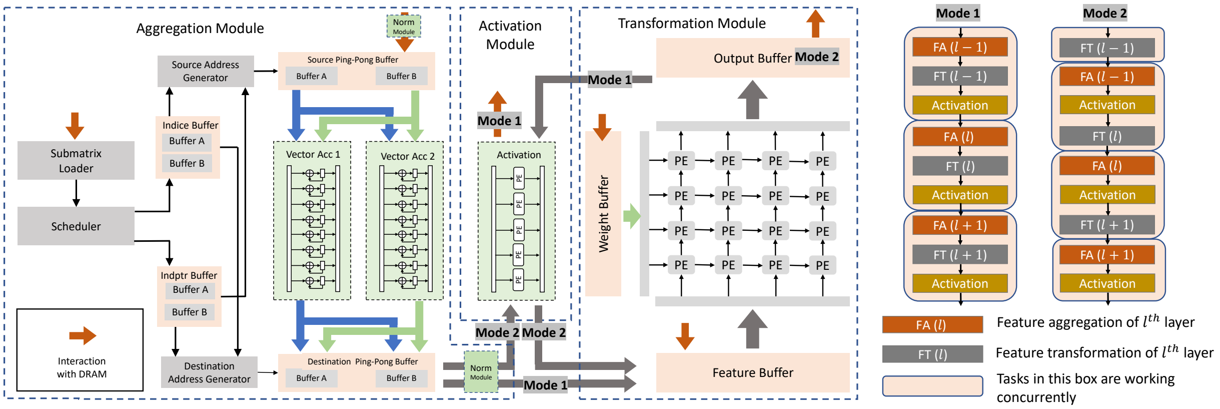


Fig. 3. Hardware Architecture with two execution modes.

operation modes (see Section V-D for details). At the same time, the Aggregation module starts to generate $\boldsymbol{C}_{i+1}^{l}$.

A. Aggregation Module

In the Aggregation module, the Submatrix Loader loads partitions $\boldsymbol{A}_{ij}$ from external memory. Each $\boldsymbol{A}_{ij}$ is stored as compressed sparse row (CSR) format. The Scheduler assigns the indice array, indptr array and value array of the submatrix into their respective buffers. The Source Address Generator and Destination Address Generator read indice and indptr to locate data in the Source Buffer (storing $\boldsymbol{X}_{j}^{l}$) and the Destination Buffer (storing $\boldsymbol{C}_{i}^{l}$). To generate $\boldsymbol{C}_{i}^{l}$, the submatrix $\boldsymbol{A}_{i0}$, $\boldsymbol{A}_{i1}$, $\boldsymbol{A}_{i2}$, ..., $\boldsymbol{A}_{i\frac{n}{k}}$ are loaded onto on-chip memory sequentially. After finishing the feature aggregation of submatrix $\boldsymbol{A}_{ij}$, the feature aggregation of $\boldsymbol{A}_{i(j+1)}$ starts. For submatrix $\boldsymbol{A}_{ij}$, feature matrix $\boldsymbol{X}_{j}^{l}$ needs to be loaded on-chip. Note that we do not need to load all the rows of $\boldsymbol{X}_{j}^{l}$. Only the rows which have corresponding non-zero element in $\boldsymbol{A}_{ij}$ need to be loaded, as shown in Figure 1. We use the double-buffering technique for hiding the memory latency. The Norm modules do element-wised multiplication for diagonal matrix (e.g. $\boldsymbol{D}^{-1}$ or $\boldsymbol{D}^{-\frac{1}{2}}$) to support the subtle variants of normalization.

To parallelize FA, we can accumulate features of multiple neighbor nodes simultaneously (*node-level* parallelism), or process multiple elements of a single neighbor feature each clock cycle (*feature-level* parallelism). Node-level parallelism may result in memory bank conflicts, such multiple nodes can access sample memory bank. Feature-level parallelism leads to regular memory accesses and can always keep the pipeline busy (since we can sequentially traverse the neighbor lists of $\boldsymbol{A}_{ij}$). However, this parallelism is limited by the feature vector length f. We exploit *both* parallelism in the Aggregation module. Note that dual-port BRAM supports two-way concurrent read and concurrent write (CRCW). The two Vector Accumulators in Figure 3 can thus read features of *any* two nodes in the same cycle. Within each Vector Accumulator, only feature-level parallelism is exploited. Therefore, the Aggregation module supports parallelism of up to $2f$, where f is the length of each feature vector. Papers [4], [8], [10], [11] shows that the feature length of GCN layer can be 128-1024 which is sufficient to support feature-level parallelism.

B. Weight Transformation Module

This module performs weight transformation of node feature vector. The input $\boldsymbol{C}_{i}^{l}$ to the Transformation module comes from the Destination Buffer fed by the Aggregation module. Since double-buffering is used in the Aggregation module, when the Transformation module accesses $\boldsymbol{C}_{i}^{l}$ from one of the Destination Buffers, the Aggregation module computes next block $\boldsymbol{C}_{i+1}^{l}$ and stores the result into the other Destination Buffer. A two dimensional systolic array is used to effectively compute the matrix multiplication between the feature matrix and weight matrix, $\boldsymbol{C}_{i}^{l}\boldsymbol{W}^{l}$. The systolic array has total parallelism of $p_{\text{sys}} \times p_{\text{sys}}$. Recall that $\boldsymbol{C}_{i}^{l} \in \mathbb{R}^{k \times f}$ and $\boldsymbol{W}^{l} \in \mathbb{R}^{f \times f}$. To fit the dimension p_{sys}, we further partition $\boldsymbol{C}_{i}^{l}$ along the rows into tiles of size $p_{\text{sys}} \times f$, and partition $\boldsymbol{W}^{l}$ along the columns into tiles of size $f \times p_{\text{sys}}$. Clearly, there are $\frac{k}{p_{\text{sys}}} \times \frac{f}{p_{\text{sys}}}$ pairs of such tiles. Each pairs requires $f + p_{sys} - 1$ clock cycles of computation. The pipelined computation of all the tile pairs takes approximately $\frac{k}{p_{\text{sys}}} \times \frac{f}{p_{\text{sys}}} \times f$ clock cycles. To stream data into the systolic array, we use a Weight Buffer of width $p_{\text{sys}} \cdot d_{\text{type}}$ (where d_{type} is the data width) to store all the weights of the GCN layer. We use a Feature Buffer of width $p_{\text{sys}} \cdot d_{\text{type}}$ bits to load the tiles of $\boldsymbol{C}_{i}^{l}$ from the Destination Buffer. The Output Buffer is a small FIFO to cache the outputs of the systolic array.

C. Scheduling for GCN variants

As indicated in Table I, there are two types of forward rules: 1) $\hat{\boldsymbol{A}}\boldsymbol{X}^{l}\boldsymbol{W}^{l}$, where $\boldsymbol{W} \in \mathbb{R}^{f_l \times f_{l+1}}$. 2) $\boldsymbol{X}\boldsymbol{W}_{\text{self}}|\hat{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}_{\text{neighbor}}$, where $\boldsymbol{W}_{\text{self}}, \boldsymbol{W}_{\text{neighbor}} \in \mathbb{R}^{f_l \times \frac{f_{l+1}}{2}}$. For the same feature length setting (f_l, f_{l+1}), the two forward rules have the same computational workload in terms of FA and FT. Using the second type, the transformation weight is divided into self weight $\boldsymbol{W}_{\text{self}}$ and neighbor weight $\boldsymbol{W}_{\text{neighbor}}$. To support the two types, we use the scheduling which indicated in Figure 4. For both of the two types, the FA and FT of the same layer are calculated concurrently. In the second type, the

FT using self weight $\boldsymbol{X}\boldsymbol{W}_{\text{self}}^{l}$ and the FT using neighbor weight $(\hat{\boldsymbol{A}}\boldsymbol{X})\boldsymbol{W}_{\text{neighbor}}$ are calculated alternatively within each data partition. The Transformation module calculates $\boldsymbol{C}_i^l\boldsymbol{W}_{\text{self}}^l$ first and then calculates $\boldsymbol{X}_i^l\boldsymbol{W}_{\text{neighbor}}^l$. Because the two types of rules have the same computational workload, under this scheduling, their execution time are the same.

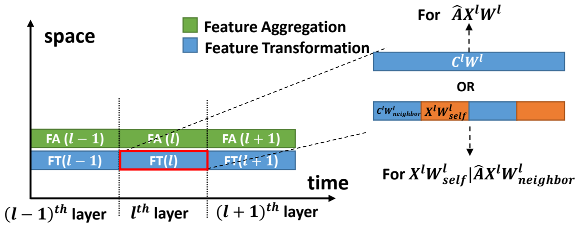


Fig. 4. Scheduling for different GCN variants.

D. Execution Modes

The product $\boldsymbol{A}\boldsymbol{X}^l\boldsymbol{W}^l$ can be computed in two different modes 1) $\left(\boldsymbol{A}\boldsymbol{X}^l\right)\boldsymbol{W}^l$ which has computation complexity of $\mathcal{O}\left(|\mathcal{E}|\cdot f^l + |\mathcal{V}|\cdot f^l \cdot f^{f+1}\right)$, or 2) $\boldsymbol{A}\left(\boldsymbol{X}^l\boldsymbol{W}^l\right)$ which has computation complexity of $\mathcal{O}\left(|\mathcal{E}|\cdot f^{l+1} + |\mathcal{V}|\cdot f^l \cdot f^{f+1}\right)$. When the next layer has shorter feature length ($f^{l+1} < f^l$), mode 2 leads to lower computation load than mode 1. When $f^{l+1} > f^l$, mode 1 is more desirable. To support the two computation orders, we implement two dataflows, as labeled by Mode 1 or Mode 2 in Figure 3. Mode 1 performs $\sigma\left(\left(\boldsymbol{A}\boldsymbol{X}^{\boldsymbol{l}}\right)\boldsymbol{W}^l\right)$. So $\boldsymbol{A}^l\boldsymbol{X}^l$ is performed first, followed by weight transformation, and further followed by activation. In Mode 2, we interleave the computation of two consecutive layers: $\left(\left(\sigma(\boldsymbol{A}\left(\boldsymbol{X}^{\boldsymbol{l}}\boldsymbol{W}^{\boldsymbol{l}}\right)\right)\boldsymbol{W}^{\boldsymbol{l+1}}\right)$. Assume that the transformation $\boldsymbol{X}^l\boldsymbol{W}^l$ of previous layer has already been performed. Then, we perform aggregation and calculate the activation of the previous layer. Next, we perform the transformation $\boldsymbol{X}^{l+1}\boldsymbol{W}^{l+1}$ of the current layer.

VI. Analysis of Computation and Communication

To simplify the performance analysis, we assume $f^l = f$, $\forall\ 1 \leqslant l \leqslant L$. L is the number of layers. So, the weight matrix $\boldsymbol{W}^l \in \mathbb{R}^{f\times f}$. We specify the hardware parameters as follows: The block size (adjacency submatrix) is k. The parallelism of a single Vector Accumulator is p_{va}. The parallelism of the systolic array is $p_{\text{sys}} \times p_{\text{sys}}$. Regarding the parameters of the graph, we define the average degree of $\mathcal{G}$ as $\bar{d}$. As for the hardware resources on FPGA, we assume the number of DSP is R_{DSP}, the external memory bandwidth is B, the frequency of the FPGA is $\mathcal{F}$. What's more, we denote the data width as d_{type}. For example, d_{type} of float32 is 4 Byte.

A. Computation

Computation time of one GCN layer equals:

$$\textcircled{1}\ T_{\text{agg}}^{\text{comp}} = \frac{n\times\bar{d}\times f}{2\times p_{\text{va}}\times\mathcal{F}} \qquad \textcircled{2}\ T_{\text{trans}}^{\text{comp}} = \frac{n\times f\times f}{p_{\text{sys}}\times p_{\text{sys}}\times\mathcal{F}} \tag{2}$$

where $2p_{\text{va}} + p_{\text{sys}}^2 \leqslant R_{\text{DSP}}$ and $p_{\text{va}} \leqslant f$. The DSP resources can be assigned based on the workload of aggregation and transformation. Ideally, if the system is bounded by computation and there is no load imbalance, we can let $T_{\text{agg}}^{\text{comp}} = T_{\text{trans}}^{\text{comp}}$ to decide the parameters p_{va} and p_{sys}. However, in practice, the aggregation time can be bounded by the external memory access, and load imbalance within the aggregation module may further affect the execution time. So, we set p_{va} 2× or 3× than needed in implementation.

B. External Memory Accesses

Aggregation module loads $\boldsymbol{X}_j^l$ from external memory into the on-chip memory, which we define as input memory accesses D_{in}. After the transformation, the results are written back to the external memory. We define D_{out} as the number of output memory accesses. For each layer, $D_{\text{out}} = n\times f\times d_{\text{type}}$. The input memory access depends on block size k and locality of $\mathcal{G}$. Here we use $\mathbb{1}_{\boldsymbol{A}_{ij}}^z$ to denote whether the z^{th} column of $\boldsymbol{A}_{ij}$ has non-zero elements. If there is any non-zero element in z^{th} column of $\boldsymbol{A}_{ij}$, $\mathbb{1}_{\boldsymbol{A}_{ij}}^z = 1$, otherwise $\mathbb{1}_{\boldsymbol{A}_{ij}}^z = 0$. Using this notation, we can drive the expression for the number of input memory accesses per GCN layer:

$$D_{\text{in}} = \sum_{i=0}^{\frac{n}{k}-1}\sum_{j=0}^{\frac{n}{k}-1}\sum_{z=0}^{k-1}\mathbb{1}_{\boldsymbol{A}_{ij}}^z\times f\times d_{\text{type}} \tag{3}$$

Here, we define a memory access factor $\mathcal{L}_i^j = \sum_{z=0}^{k-1}\mathbb{1}_{\boldsymbol{A}_{ij}}^z$ to denote the input memory access for each submatrix $\boldsymbol{A}_{ij}$. Let $\mathcal{L} = \sum_{i=0}^{\frac{n}{k}-1}\sum_{j=0}^{\frac{n}{k}-1}\mathcal{L}_i^j$. Using D_{in} and D_{out}, we can drive the expression of communication time: $T^{\text{comm}} = \frac{D_{\text{in}}+D_{\text{out}}}{B}$ which denotes the time for memory access given the memory bandwidth. Using practical graph and GCN models, the performance of the system can be bounded by the data communications (Section VII-D). So, given an input graph, the two-phase algorithm optimization can significantly reduce the factor $\mathcal{L}$. As a result, communication time T^{comm} is reduced. Here, we do not consider the memory traffic of loading weights ($D_{\text{weight}} = f^2$, $f << n << \mathcal{L}$) because it is negligible compared with the memory traffic of loading input features and output features.

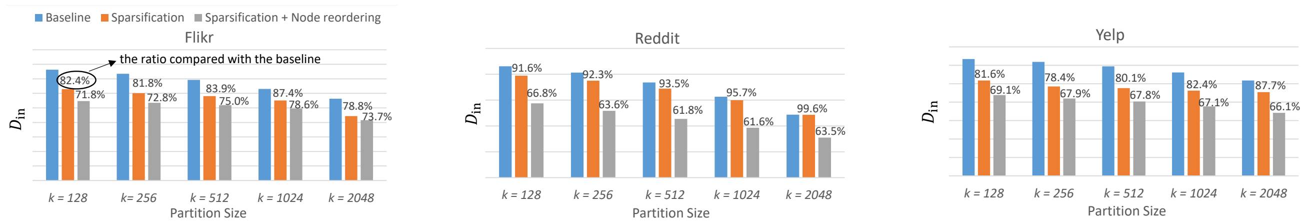
VII. Evaluation

A. Experimental Setting

We implement our design in verilog-HDL and use the Xilinx Alveo U200 accelerator card as the platform for evaluation. The acceleration card has 64 GB off-chip DRAM (77 GB/s bandwidth), 1182k Look-up tables (LUTs), 6840 DSPs, 2160 36k BRAM and 960 288k Ultra RAM. We use the Float32 as the default word length to represent weights and features. Vivado 2018.03 is used for synthesis [1].

For evaluation purpose, we use the three widely used datasets Reddit, Yelp, Flickr [10], [21], [4], [22] as the input graph. The details of the datasets are shown in Table II. We store all the data in the off-chip DRAM. The node order of the original dataset is randomized in order to evaluate the

[1] https://github.com/GraphSAINT/GNN-ARCH



(a) (b) (c)

Fig. 5. D_{in} of Flikr (a), Reddit (b), Yelp (c) for various partition size k.

TABLE III

CROSS PLATFORM COMPARISON OF THROUGHPUT (MENPS). THERE ARE TWO NUMBERS IN EACH CELL OF THE TABLE. THE NUMBER ON THE LEFT IS THE THROUGHPUT USING THE UN-PRE-PROCESSED DATASET. THE NUMBER ON THE RIGHT IS THE THROUGHPUT USING THE DATASET AFTER TWO-PHASE PRE-PROCESSING.

	Dataset	**Tensorflow**		**Hand-written code**		**Xiline Alveo U200** ($k = 1024$)
		CPU (56 threads)	GPU (Titan-Xp)	CPU (56 threads)	GPU (Titan-Xp)	
$f = 128$	Flikr	0.213 / 0.226	0.829 / 0.844	0.129 / 0.159	2.985 / 3.030	2.076 / **2.452** ($p_{va} = 128, p_{sys} = 24$)
	Reddit	0.032 / 0.034	0.116 / 0.120	0.096 / 0.099	1.253 / 1.294	1.021 / **1.641** ($p_{va} = 128, p_{sys} = 24$)
	Yelp	0.112 / 0.121	0.338 / 0.349	0.157 / 0.168	2.542 / 2.610	2.743 / **3.854** ($p_{va} = 128, p_{sys} = 24$)
$f = 256$	Flikr	0.139 / 0.147	0.702 / 0.708	0.048 / 0.051	0.572 / 0.577	0.941 / **1.065** ($p_{va} = 64, p_{sys} = 24$)
	Reddit	0.018 / 0.016	0.086 / 0.087	0.037 / 0.039	0.434 / 0.440	0.506 / **0.815** ($p_{va} = 256, p_{sys} = 24$)
	Yelp	0.091 / 0.097	0.317 / 0.323	0.050 / 0.052	0.642 / 0.648	0.711 / **0.956** ($p_{va} = 128, p_{sys} = 24$)

TABLE IV

THE IMPACT OF TWO-PHASE PREPROCESSING ALGORITHM ON THE THROUGHPUT (MENPS).

	Feature Length	**Hardware Configuration**	**Baseline**	**Sparsi.**	**Sparsi. + Rordering**
Flikr	$f = 128$	$p_{va} = 128$ $p_{sys} = 24$	2.076	2.336	2.452
	$f = 256$	$p_{va} = 64$ $p_{sys} = 24$	1.021	1.050	1.065
Reddit	$f = 128$	$p_{va} = 128$ $p_{sys} = 24$	1.021	1.059	1.641
	$f = 256$	$p_{va} = 256$ $p_{sys} = 24$	0.506	0.529	0.815
Yelp	$f = 128$	$p_{va} = 128$ $p_{sys} = 24$	2.743	2.914	3.854
	$f = 256$	$p_{va} = 128$ $p_{sys} = 24$	0.711	0.747	0.956

impact of our pre-processing algorithm. The implementation of Reverse Cuthill-McKee algorithm uses the build-in function in *scipy*. For all the three datasets, following the setting in papers [10], [21], [4], [22], we use a two-layer GCN model. The feature length is set as $f^l = 128, 256$ $(0 \leqslant l < L)$ and $L = 2$. We use the forward rule $\hat{\boldsymbol{A}}\boldsymbol{X}^l\boldsymbol{W}^l$, since $\hat{\boldsymbol{A}}\boldsymbol{X}^l\boldsymbol{W}^l$ and $\boldsymbol{X}\boldsymbol{W}_{\text{self}}|\hat{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}_{\text{neighbor}}$ have the same execution time using our scheduling method (Section V-C). We do not include the time for data pre-possessing as explained in Section III and IV-D.

B. External Memory Accesses

Since $\mathcal{L} >> n$, D_{out} is negligible compared with D_{in}, D_{out} is ignored for evaluation. D_{in} is related to the block size k and data locality. The experimental results are shown in Figure 5. As k increases, external memory access decreases. Using larger block size k, larger submatrix can be loaded on-chip, leading to the increased data reuse. Suppose k is large enough to store the entire graph, the input graph only needs to be loaded once. Using the two-phase pre-processing algorithm, the first sparsification phase can reduce the external memory access by 17.5%-21.2%, 1%-8%, 12.3%-18.4% for Flikr, Reddit, Yelp respectively. Incorporating the node reordering, the external memory access can be further reduced by 26.3%-28.2%, 33.2%-39%, 30.9%-33.9% for Flikr, Reddit, Yelp respectively. The experimental results confirm that our two-phase preprocessing algorithm can effectively reduce the external memory access, which can alleviate the bottleneck of memory bandwidth.

C. Cross-platform Comparison

We compare our FPGA implementation with the baseline implementations using Tensorflow (version 2.0.0) and our optimized hand-written C++ code. Both of the baseline implementations have two versions running on multi-core CPU platform with 56 threads (Intel Xeon 5120 @ 2.20 GHZ) and high end GPU platform (Titan-Xp) respectively. For multi-core CPU platform, Tensoflow CPU version is used and our hand-written C++ code uses the Pthread library. For GPU platform, Tensorflow GPU version is used and our hand-written C++ code uses Cuda 10.0 library. For cross-platform comparison, we use the unpre-processed dataset and the pre-processed dataset using two-phase algorithm. For FPGA implementation, the block size k is set as 1024 to fit the FPGA on-chip memory. Each processing element in vector Accumulator consumes 5

DSPs and each processing element in systolic array consumes 7 DSPs. The frequency can reach nearly 300 MHZ and we set 250 MHZ for the implementation.

The comparsion results are shown in Table III. Compared with the two CPU baselines, we achieve 10× to 30× higher throughput. Compared with the GPU implementation, we achieve 2× to 10× speed. Compared with CPU and GPU implementation, our FPGA implementation is more efficient because 1) FPGA has larger on-chip memory and it can be used to store larger data partition, which leads to more data reuse. Due to the capacity to store larger data partition, FPGA implementation benefits more from the two-phase pre-processing algorithm. CPU and GPU implementations mainly benefit from graph sparsification, which eliminates the edge connections. 2) The communication between two computational kernels–FA and FT–are through the FPGA on-chip memory which is more efficient than communicating through external memory.

D. Impact of Two-phase Algorithm

In Table IV, we show the impact of two-phase preprocessing algorithm on the inference throughput (MNEPS). The baseline uses the input graph without pre-processing. We measure the throughput after each phase of the pre-processing. With proposed algorithm, we improve the throughput by 15%, 37.7%, 29% on Flikr, Reddit, Yelp respectively. The increased performance is because aggregation has intensive external memory access which limits the performance. Using the two-phase pre-processing, we increase the data reuse in each partition $\boldsymbol{A}_{ij}$ which greatly reduce the external memory access.

In our experiments, we notice that memory bottleneck is more severe for input graphs with high average degree like Reddit. Because with more average degree, more external memory access is needed for feature aggregation. In practice, graphs with high average degree is common. For example, in social networks, users may have hundreds of links with other users on the average. In e-commerce system [6], customers may go through hundreds of products on the average. Our two-phase preprocessing algorithm can potentially improve the performance caused by the memory traffic bottleneck.

VIII. Conclusion

In this paper, we proposed the algorithm-architecture co-optimization to accelerate GCN inference on FPGA. The hardware used pipelined implementation of aggregation and transformation to achieve high-throughput inference. The two-phase preprocessing algorithm further improved the throughput by optimizing the external memory access. In the future, we intend to apply our proposed method to other types of graph neural network.

Acknowledgment

This work was partially supported by the US NSF under grant No. CCF-1919289 and No. OAC-1911229, Intel Strategic Research Alliance and the Defense Advanced Research Projects Agency under contract No. FA8750-17-C-0086.

References

[1] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *arXiv:1901.00596*, 2019.

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

[3] M. Zitnik and J. Leskovec, "Predicting multicellular function through multi-layer tissue networks," *Bioinformatics*, 2017.

[4] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 1024–1034.

[5] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," *CoRR*, vol. abs/1806.01973, 2018.

[6] 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*, vol. 12, no. 12, 2019.

[7] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting," *arXiv preprint arXiv:1707.01926*, 2017.

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

[9] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V.P., "Accurate, efficient and scalable graph embedding," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 462–471.

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

[11] 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," *arXiv preprint arXiv:1905.07953*, 2019.

[12] Y. Meng, S. Kuppannagari, and V. Prasanna, "Accelerating proximal policy optimization on cpu-fpga heterogeneous platforms," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 19–27.

[13] B. Zhang, J. Han, Z. Huang, J. Yang, and X. Zeng, "A real-time and hardware-efficient processor for skeleton-based action recognition with lightweight convolutional neural network," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 66, no. 12, pp. 2052–2056, 2019.

[14] H. Zeng and V. Prasanna, "Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 255–265.

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

[16] X. Geng, Y. Li, L. Wang, L. Zhang, Q. Yang, J. Ye, and Y. Liu, "Spatiotemporal multi-graph convolution network for ride-hailing demand forecasting," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[17] 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, 2019.

[18] T. Geng, A. Li, T. Wang, C. Wu, Y. Li, A. Tumeo, and M. Herbordt, "Uwb-gcn: Hardware acceleration of graph-convolution-network through runtime workload rebalancing," *arXiv preprint arXiv:1908.10834*, 2019.

[19] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, 2013.

[20] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th national conference*. ACM, 1969, pp. 157–172.

[21] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," *arXiv preprint:1710.10568*, 2017.

[22] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive sampling towards fast graph representation learning," in *Advances in Neural Information Processing Systems*, 2018, pp. 4558–4567.