BoostGCN: A Framework for Optimizing GCN Inference on FPGA

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

***Abstract*—Graph convolutional networks (GCNs) have revolutionized many big data applications, such as recommendation systems, traffic prediction, etc. However, accelerating GCN inference is challenging due to (1) massive external memory traffic and irregular memory access, (2) workload imbalance due to skewed degree distribution, and (3) intra-stage load imbalance caused by two heterogeneous computation phases of the algorithm. To address the above challenges, we propose a framework named BoostGCN to optimize GCN inference on FPGA. First, we develop a novel hardware-aware Partition-Centric Feature Aggregation (PCFA) scheme that leverages 3-D partitioning with the vertex-centric computing paradigm. This increases on-chip data reuse and reduces the total data communication volume with external memory. Second, we design a novel hardware architecture to enable pipelined execution of the two heterogeneous computation phases. We develop a low-overhead task scheduling strategy to reduce the pipeline stalls caused by the two computation phases. Third, we provide a complete GCN acceleration framework on FPGA with optimized RTL templates. It can generate hardware designs based on the customized configuration and is adaptable to various GCN models. Using our framework, we generate accelerators for various GCN models on a state-of-the-art FPGA platform and evaluate our designs using widely used datasets. Experimental results show that the accelerators produced by our framework achieve significant speedup compared with state-of-the-art implementations on CPU** $(\approx 100\times)$**, GPU** $(\approx 30\times)$**, prior FPGA accelerator** $(3\text{-}45)\times$**.**

I. Introduction

Graph convolutional networks (GCNs) have become a revolutionary machine learning technology in many big data applications where data are represented as graphs. For example, in recommendation systems (e.g. Aligraph [1], Pinterest [2]), GCN is used for personalized recommendation. Smart transportation systems [3], [4], [5] exploit GCN to perform real-time traffic prediction. In Chemistry, GCN is used to learn and predict the properties of proteins [6], [7].

GCN operates on vertex-attributed graphs, where each vertex is attributed by a feature vector. GCN contains a stack of GCN layers [6] with each layer following the aggregate-update paradigm. In feature aggregation phase, each vertex aggregates features from its neighbors and performs reduction. In feature update phase, each vertex transforms the aggregated new feature vector to another feature space. For example, the combination of Multi-layer Perceptron (MLP) and element-wise activation function (e.g. ReLU, LeakyReLU) is commonly used to transform feature vectors.

Despite the popularity of GCN, accelerating GCN inference is still challenging, due to (1) extensive memory traffic and irregular memory accesses in feature aggregation, (2) workload imbalance by the skewed degree distribution, and (3) intra-stage load imbalance between the aggregation phase and update phase. The aggregation phase involves memory-intensive graph traversal while the update phase involves computation-intensive feature transformation. The heterogeneous computation pattern can result in stalled execution.

In this paper, we develop a framework named BoostGCN to optimize GCN inference on FPGAs. First, we propose a novel hardware-aware Partition-Centric Feature Aggregation (PCFA) scheme to improve the execution efficiency of feature aggregation. To facilitate PCFA, we design the novel accelerator with a corresponding task scheduling strategy to reduce the pipeline stalls of the two computation phases. Our proposed framework achieves high-throughput, at the same time maintaining good adaptability to various GCN models. Our framework comprehensively deals with the inefficient memory accesses of GCN feature aggregation and adapts to the variability of GCN models. Our main contributions are:

- We propose a hardware-aware Partition-Centric Feature Aggregation (PCFA) scheme. PCFA performs 3-D graph partitioning by considering on-chip data reuse as well as the architectural characteristics of the external memory. This reduces memory traffic and overall memory access latency. PCFA also addresses workload imbalance by using a centralized load balancing scheme.
- We design a novel accelerator to facilitate our PCFA scheme and achieve massive data parallelism. The proposed accelerator enables pipelined execution of the two heterogeneous computation phases. We propose a task scheduling strategy to reduce the pipeline stalls of the two computation phases.
- We integrate our PCFA scheme and hardware architecture as a complete framework with optimized RTL templates. Our framework can generate accelerators based the customized configurations. Thus it can be quickly adapted to various GCN models.
- Using our framework, we generate accelerators on a state-of-the-art FPGA platform and evaluate our designs using widely used datasets. Compared with CPU (PyG-CPU [8]/DGL-CPU [9]) and GPU (PyG-GPU/DGL-GPU), we

TABLE I
GCN 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	$\boldsymbol{A}$	adjacency matrix
N	number of vertices	nnz	number of edges
$\boldsymbol{X^0} \in \mathbb{R}^{N \times f_0}$	feature matrix of $\mathcal{V}$	f_0	feature length of $\mathcal{G}$
L	number of GCN layers	f_l	feature length of layer l
h_i^l	feature vector of v_i	$\overline{\boldsymbol{A}}$	$\overline{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$
$\boldsymbol{X}^l \in \mathbb{R}^{N \times f_l}$	feature matrix of layer l	$\mathcal{N}(i)$	neighbors of v_i
$\phi(\cdot)$	element-wise activation	$(\|)$	row concatenation

achieve nearly $100\times$ and $30\times$ speedups respectively. Compared with a state-of-the-art FPGA accelerator [10], we achieve $(3\text{-}45)\times$ speedup.

II. BACKGROUND

A. Notations

We define the related notations in Table I. We use bold capital letter (e.g. $\boldsymbol{X}$) to denote matrix. $\boldsymbol{X}_i$ denotes the i^{th} row of $\boldsymbol{X}$, and $\boldsymbol{X}_{:,j}$ denotes the j^{th} column of $\boldsymbol{X}$. $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X^0})$ denotes the input graph to the GCN model, where $\mathcal{V}$ denotes the set of vertices and $\mathcal{E}$ denotes the set of edges. v_i denotes the i^{th} vertex of $\mathcal{G}$. e_{ij} denotes the edge from v_i to v_j. N denotes the number of vertices. $\boldsymbol{A}$ is the adjacency matrix of the graph. $\boldsymbol{X^0}$ denotes the feature matrix of $\mathcal{G}$ where $\boldsymbol{X_i^0}$ is the attributed feature vector to v_i. We use $\boldsymbol{X_i^0}[a:b]$ to denote a slice of the feature vector from a^{th} feature to b^{th} feature. We use coordinate format (COO) to represent sparse matrix.

B. Graph convolutional networks

Algorithm 1 Aggregate-update paradigm of GCN

Input: Initial vertex features $(h_1^0, h_2^0, h_3^0, ..., h_N^0)$
Output: Final vertex embeddings $(h_1^L, h_2^L, h_3^L, ..., h_N^L)$
1: **for** $l \leftarrow 1$ to L **do**
2: **for** $i \leftarrow 1$ to N **do**
3: z_i^{l-1} = aggregate($h_j^{l-1} : j \in \mathcal{N}(i) \bigcup \{i\}$)
4: h_i^l = update(z_i^{l-1})
5: **end for**
6: **end for**

GCNs were proposed [6] to learn the representations of vertices in graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X^0})$, by taking both graph topology and the vertex features into account. The original feature vector attached to v_i is $h_i^0 \in \mathbb{R}^{f_0}$ ($h_i^0 = \boldsymbol{X_i^0}$). After L GCN-layer operations, v_i gets the vector representation $h_i^L \in \mathbb{R}^{f_L}$. The obtained vector representation can be used for many down-stream tasks, such as node classification [11], link prediction [12], etc. Each GCN layer follows the aggregate-update paradigm as shown in Algorithm 1. **Aggregate:** in l^{th} layer, v_i aggregates features from its neighbors and uses the algorithm-specific operators (e.g. add, max, min [11]) to reduce the aggregated features to a new feature vector z_i^l. **Update:** the aggregated feature vector z_i^{l-1} is transformed to h_i^l by the combination of linear transformation (e.g. MLP) and non-linear transformation (e.g. ReLU). While there are many

TABLE II
DATASET STATISTICS

Graph	Matrix	Size	Density	Graph	Matrix	Size	Density
Cora	$\boldsymbol{A}$	$2.7K \times 2.7K$	0.14%	Flickr	$\boldsymbol{A}$	$89K \times 89K$	0.011%
	$\boldsymbol{X}^0$	$2.7K \times 1.4K$	1.27%		$\boldsymbol{X}^0$	$89K \times 500$	46.3%
Citeseer	$\boldsymbol{A}$	$3.3K \times 3.3K$	0.08%	Reddit	$\boldsymbol{A}$	$0.23M \times 0.23M$	0.04%
	$\boldsymbol{X}^0$	$3.3K \times 3.7K$	0.85%		$\boldsymbol{X}^0$	$0.23M \times 602$	51.6%
Pubmed	$\boldsymbol{A}$	$19K \times 19K$	0.023%	Yelp	$\boldsymbol{A}$	$0.72M \times 0.72M$	0.0027%
	$\boldsymbol{X}^0$	$19K \times 500$	10.02%		$\boldsymbol{X}^0$	$0.72M \times 300$	50.9%
PPI	$\boldsymbol{A}$	$14K \times 14K$	0.21%	Amazon	$\boldsymbol{A}$	$1.6M \times 1.6M$	0.011%
	$\boldsymbol{X}^0$	$14K \times 50$	1.8%		$\boldsymbol{X}^0$	$1.6M \times 200$	49.7%

advanced GCN models, such as Vanilla-GCN [6], GraphSage [11], GraphSAINT [13], GINconv [14], etc, their main difference is the training method and the model-specific operators. For inference, they have the same computation kernels within feature aggregation and feature update. Without losing generality, we use Vanilla-GCN for illustration in this paper. And our framework is also applied to other GCN models.

Vanilla-GCN: Vanilla-GCN [6] is the first proposed GCN model to learn the vertex representations in a graph. The vectorized representation of aggregate-update paradigm is denoted as:

$$\boldsymbol{X}^l = \text{ReLU}\left(\tilde{\boldsymbol{A}}\boldsymbol{X}^{l-1}\boldsymbol{W}^l\right), \tag{1}$$

where $\tilde{\boldsymbol{A}} = \boldsymbol{D}^{-\frac{1}{2}}\overline{\boldsymbol{A}}\boldsymbol{D}^{-\frac{1}{2}}$. $\overline{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$ is the self-loop adjacency matrix. $\boldsymbol{D}$ is Laplacian matrix with $\boldsymbol{D}_{ii} = \sum_j \overline{\boldsymbol{A}}_{ij}$. $\boldsymbol{W}^l \in \mathbb{R}^{f_{l-1} \times f_l}$ is the weight matrix of l^{th} layer. The commonly used two-layer GCN model is represented as:

$$\boldsymbol{X}^2 = ReLU\left(\tilde{\boldsymbol{A}}\ ReLU\left(\tilde{\boldsymbol{A}}\boldsymbol{X}^0\boldsymbol{W}^1\right)\boldsymbol{W}^2\right). \tag{2}$$

TABLE III
FEATURE LENGTH IN GCN MODELS

Dataset	f_0	f_1	f_2	Dataset	f_0	f_1	f_2
Cora	1433	16/128	7	Flickr	500	128/256	7
Citeseer	3703	16/128	6	Reddit	602	128/256	41
Pubmed	500	16/128	3	Yelp	300	128/256	100
PPI	50	128/256	121	Amazon	200	128/256	107

C. Challenges

To understand the challenges within GCN acceleration, we summarize the widely used graphs from [15], [11], [6], [16], [13], [10], in Table II. We identify several challenges in accelerating GCN:

- **Challenge 1:** Adjacency matrix has extremely low density and uneven distribution of non-zero elements, which results in low data reuse and random memory access in feature aggregation.
- **Challenge 2:** Graphs have extremely skewed degree distribution. Therefore, different vertices have different workloads, leading to workload imbalance within feature aggregation.
- **Challenge 3:** While the feature aggregation is communication-intensive and feature update is computation-intensive. The heterogeneous computation patterns can potentially lead to stalled pipeline execution.

- **Challenge 4:** Graphs have various size and sparsity (Table II), and the GCN model has various parameters (Table III). A GCN framework needs to adapt to the variations.

III. Related Work

CNN Accelerators: Many CNN (convolutional neural network) accelerators [17], [18], [19], [20], [21], [22] have been proposed to achieve high-throughput CNN inference. However, their techniques do not generalize to GCNs, which have poor data locality and irregular memory accesses. Another type of CNN accelerators [23], [24], [25] explores the data sparsity within weight or activation of CNN layers to reduce the inference latency. However, they are unsuitable for GCNs because (1) the sparse matrix in GCNs is significantly larger than CNN weight matrix or activation matrix, (2) the sparsity of matrices (99%) in GCNs is significantly higher than the sparsity of matrix (10%-50%) in CNN, and (3) GCNs have highly unbalanced degree distribution.

Graph Processing Accelerators: Graph Processing Accelerators deal with the graph traversal problems with each vertex attributed by a single scalar, such as PageRank. The representative graph processing accelerators using edge-centric paradigm include Hitgraph [26] and [27]. However, applying the two-phase edge-centric paradigm to GCN feature aggregation leads to extensive memory traffic. Another type of graph processing accelerators uses vertex-centric paradigm to reduce memory traffic, including FPGP [28], Foregraph [29], Fabgraph [30]. Nevertheless, they are exclusively optimized for the graph traversal with each vertex attributed by a scalar. Applying them to GCN feature aggregation, memory traffic of loading edges becomes the bottleneck. Thus, they are unsuitable for GCNs without significant changes.

GNN Accelerators: Many works have been proposed to accelerate GCN. [13] introduces a fast GCN training algorithm GraphSAINT, and implement it on a CPU platform [31] and a CPU-FPGA heterogeneous platform [32]. In this paper, we focus on GCN inference, which is also the focus of recent works [33], [34], [16], [10], [15]. HyGCN [16] proposes a hybrid accelerator for GCN inference by exploiting ASIC technology. It uses a 2-D graph partitioning strategy to improve the data locality. In our previous work [10], we developed an algorithm-architecture co-design framework for GCN inference on FPGA. We developed redundancy reduction and node reordering techniques to increase data locality and reduce redundant computations in feature aggregation. However, previous works ([16], [10]) do not consider the graph partitioning along the feature dimension. This still leads to large memory traffic (See Section IV-B).

IV. System Architecture

In this section, we introduce the architecture of BoostGCN, followed by the architecture of the Feature Aggregation Module with the mechanism of partition-centric feature aggregation. Then we explain the Feature Update Module and task scheduling in detail.

A. Overall Architecture

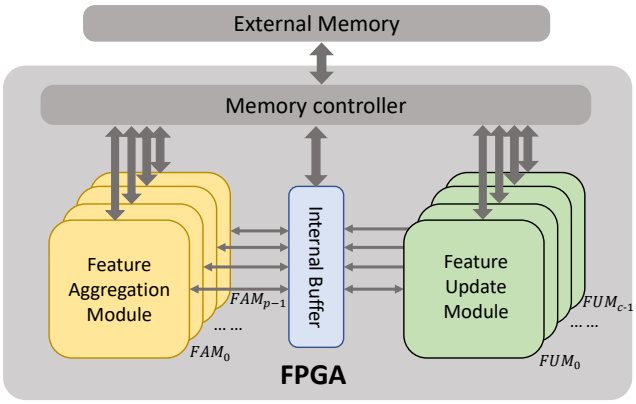

Fig. 1. Overall architecture of BoostGCN.

The overall system architecture produced by BoostGCN is shown in Figure 1, consisting of external memory and FPGA. The external memory stores adjacency matrix $\tilde{\boldsymbol{A}}$, feature matrices $\boldsymbol{X^0}, \boldsymbol{X^1}, ..., \boldsymbol{X^L}$, and weight matrices $\boldsymbol{W^1}, \boldsymbol{W^2}, ..., \boldsymbol{W^L}$. On the FPGA board, Feature Aggregation Modules (FAMs) perform feature aggregation $\tilde{\boldsymbol{A}}\boldsymbol{X}$, and Feature update modules (FUMs) perform feature update $(\tilde{\boldsymbol{A}}\boldsymbol{X})\boldsymbol{W}$. The Internal Buffer caches the intermediate results $(\tilde{\boldsymbol{A}}\boldsymbol{X})$ produced by FAMs. Memory controller handles data transmissions between external memory and hardware modules.

B. Partition-Centric Feature Aggregation (PCFA)

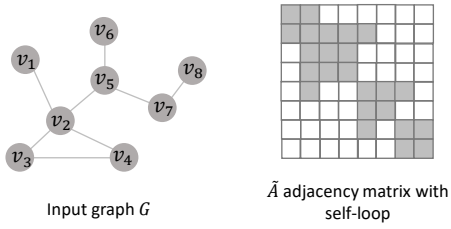

Fig. 2. An example of input graph with corresponding adjacency matrix.

Feature aggregation $\tilde{\boldsymbol{A}}\boldsymbol{X}$ is performed by FAMs. $\tilde{\boldsymbol{A}}$ contains the edges within the input graph. In COO format, each edge is represented as a three-element tuple $(src, dst, value)$ to specify the source vertex, destination vertex, and edge weight, respectively. In GCN, feature aggregation imposes great challenges for acceleration due to extensive memory traffic and low bandwidth utilization. Thus, it contributes to the most part of execution time within GCN-layer operation [16], [15]. The memory traffic contains two parts: (1) loading edges (2) loading/storing vertex features. The well-known vertex-centric paradigm can save memory traffic of graph traversal with some enhancements such as graph tiling and optimized data replacement strategy [28], [29], [30]. However, these optimizations are for graph traversal problems with each

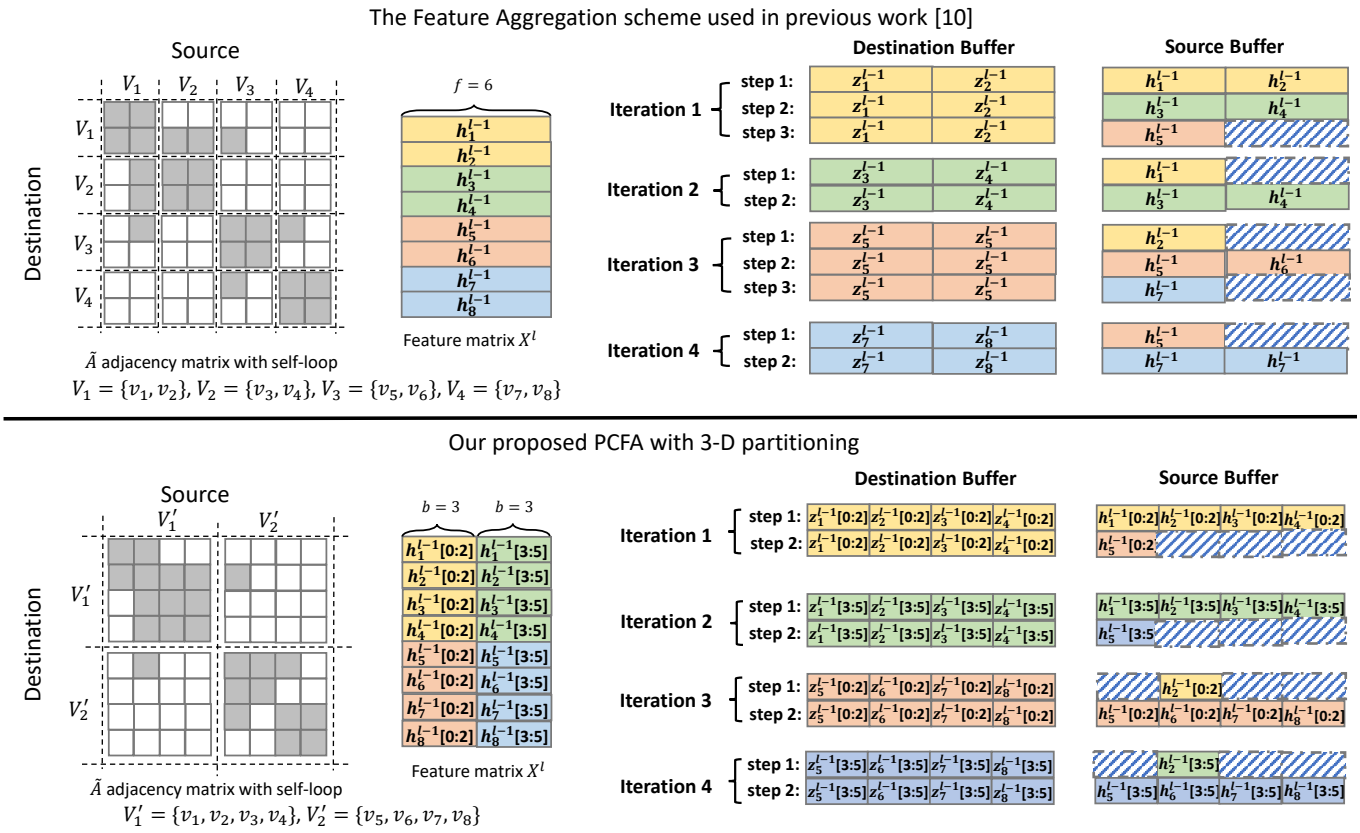

Fig. 3. Proposed Partition-Centric Feature Aggregation scheme (PCFA). Figure 2 shows the input graph for this example.

vertex attributed by a scalar. Thus they are inefficient for GCN feature aggregation. Previous GCN accelerators [16] [10] extend vertex-centric paradigm to GCN feature aggregation as depicted in the upper part of Figure 3. However, such direct extension is not efficient for GCN acceleration. (1) Due to the feature dimension ($f_i, 0 \leqslant i \leqslant L$), on-chip memory can only store a limited number of vertices, leading to poor on-chip data reuse. (2) Due to the feature dimension, feature vectors are likely to span different DRAM pages, potentially leading to large number of DRAM row misses or row conflicts. (3) Prior work do not consider the burst length of data transaction, which is an architectural parameter of external memory, leading to memory bandwidth wastage. Thus, previous works did not optimize the on-chip data reuse by graph partitioning along the feature dimension, or consider architectural characteristics of external memory.

Architecture-aware optimizations are key to improve the performance of graph processing. Therefore, we propose the PCFA scheme which is an algorithm-architecture co-optimization that improves memory performance of GCN feature aggregation as well as achieves massive computation parallelism.

Data tiling: In PCFA, we perform data tiling in three dimensions: (1) vertices, (2) edges, (3) vertex features. The vertex set $\mathcal{V}$ is divided into disjoint subsets denoted intervals. Each interval V_i contains k vertices. The edge set $\mathcal{E}$ is divided

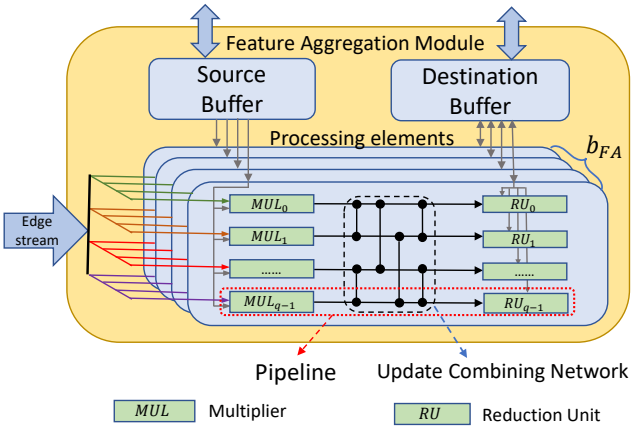

Fig. 4. The architecture of FAM (Feature Aggregation Module).

into subsets named blocks. Block $E_{i \to j}$ contains the edges which have source vertex in V_i and destination vertex in V_j ($0 \leqslant i, j \leqslant \frac{N}{k} - 1$). Therefore, $\bigcup_{j=0}^{\frac{N}{k}-1} E_{j \to i}$ contains all the incoming edges of V_i. Then, we perform data partitioning along feature dimension (3-D partitioning) for each vertex. The feature dimension is divided into slices by 3-D partitioning. Each slice contains consecutive b features. The lower part of Figure 3 depicts an example of data tiling of $\tilde{\boldsymbol{A}}$ and $\boldsymbol{X}$ under PCFA. A data partition contains an interval V_i with k vertices, all the incoming edges of V_i that is $\bigcup_{j=0}^{\frac{N}{k}-1} E_{j \to i}$, and

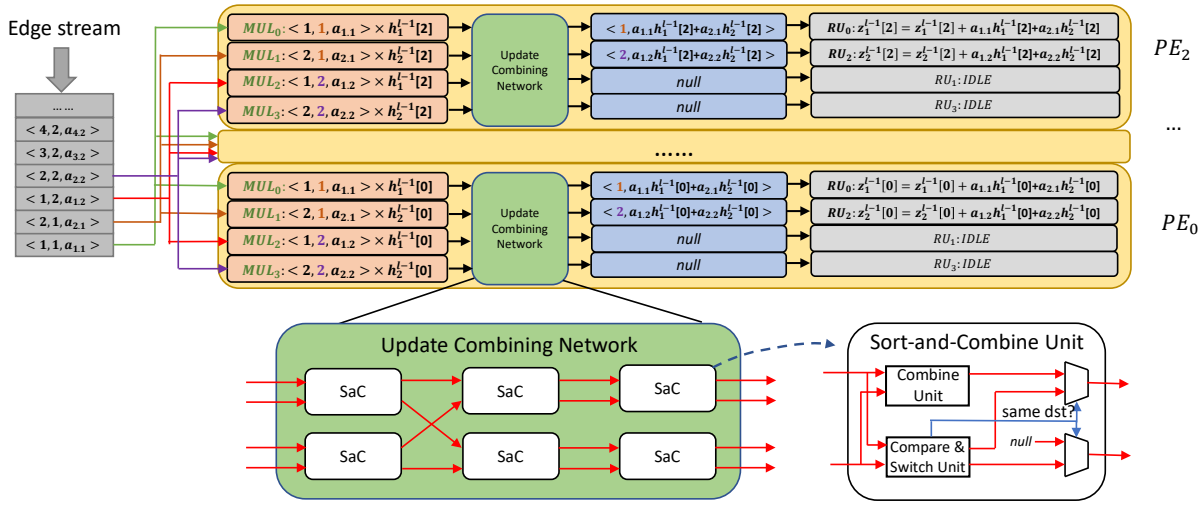

Fig. 5. An example of computation process in FAM.

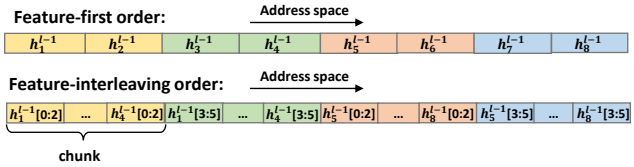

Fig. 6. Feature-interleaving order of feature matrix $\boldsymbol{X}$.

b features of each vertex in V_i that is $\{X_v[hb:(h+1)b-1]: v \in V_i\}, (0 \leqslant h \leqslant \frac{f}{b}-1)$. We use an index-based partition scheme having the complexity of $O(|\mathcal{E}|)$. Previous works [16] [10] require time-consuming graph preprocessing.

On-chip data flow: Based on the proposed data tiling strategy, FAM (Figure 4) performs feature aggregation. FAM has a source buffer to store the features of source vertices, and a destination buffer to store the features of destination vertices. The edges are streaming into FAM from external memory. We use the example in Figure 3 to illustrate the data flow, where the partition size is $k=4, b=3$. The feature aggregation is an iterative process. At 1st iteration, $z_i^{l-1}[0:2](i=1,2,3,4)$ are allocated in the destination buffer. In step 1, $h_i^{l-1}[0:2](i=1,2,3,4)$ are loaded from external memory to source buffer, and perform aggregation to get the partial results of $z_i^{l-1}[0:2](i=1,2,3,4)$. In step 2, $h_5^{l-1}[0:2]$ are loaded to source buffer. FAM performs aggregation and accumulates the incremental values to $z_i^{l-1}[0:2](i=1,2,3,4)$. Then, the aggregation of $z_i^{l-1}[0:2](i=1,2,3,4)$ is completed and sent back to internal buffer for feature update. At 2^{nd} iteration, FAM performs aggregation for $z_i^{l-1}[3:5](i=1,2,3,4)$, finishing the feature aggregation of interval V_1. The same feature aggregation process is applied to interval V_2. Compared with feature aggregation scheme used in the previous works [10], [16], PCFA has several benefits as indicated in Figure 3: (1) The increased edge block size leads to more data reuse within edge blocks, thus less data traffic with external memory. (2) Due to the increased edge block size, we can potential have less computation steps. (3) PCFA facilitates our proposed off-chip data storage scheme to reduce the overall memory access latency, and our proposed hardware architecture to achieve massive data parallelism.

Off-chip data storage: Based on the proposed data tiling strategy, we develop a data storage scheme in external memory. First, the edges are stored by blocks. In COO format, each edge is stored as a three-element tuple $(src, dst, value)$. Each element has 4 Bytes with total 12 Bytes to store an edge. Under PCFA, src and dst are used to index the Source Buffer and Destination Buffer. Therefore, src and dst can be compressed to $\log_2(k)$ bits. In BoostGCN, we use 2 Bytes to represent each of src and dst. Each edge consumes 8 Bytes storage space. Therefore, the memory traffic of loading edges is reduced by $1/3$. Second, the feature matrix $\boldsymbol{X}$ is originally stored in feature-first order. Under PCFA, we store $\boldsymbol{X}$ as feature-interleaving order (FIO), as shown in Figure 6. Using FIO, accessing features within a 3-D partition can be confined into a small range of memory space, which is in small number of several consecutive DRAM pages. Under such arrangement of feature matrix $\boldsymbol{X}$, we can potentially reduce the number of DRAM row misses and row conflicts.

Computation parallelism: The architecture of FAM facilitated by PCFA is shown in Figure 4. A FAM can process q edges in each clock cycle. There are b_{FA} processing elements (PEs) within a FAM to process b_{FA} feature dimensions. An example of the computation process in FAM is illustrated in Figure 5. There are p FAMs in the system. Therefore, the total computation parallelism of feature aggregation is $p \times b_{\text{FA}} \times q$.

Update Combining Network: The architecture of the Update Combining Network is depicted in Figure 4. A Sort-and-Combine (SaC) unit sorts the incoming intermediate results by their indices of destination vertex. If the two inputs have the same destination, the SaC unit will combine them. Using the Update Combining network, we can eliminate the congestion when intermediate results have the same destination vertex

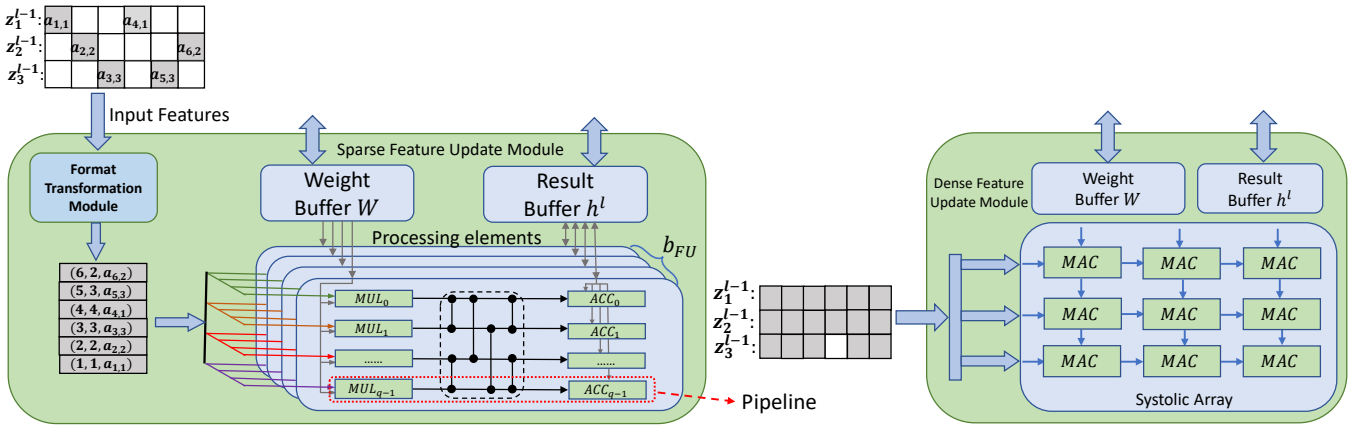

Fig. 7. The architecture of two types of Feature Update Modules.

index.

Double buffering: We exploit the double buffering technique for Source Buffer and Destination Buffer, so that the computation and memory access can be overlapped to improve the overall throughput.

Partition size: In PCFA, partition size k, b are selected by considering (1) the trade-off between feature partition size b and vertex partition size k to minimize external memory traffic, and (2) the architectural characteristics of external memory to minimize the bandwidth wastage. Suppose N_{SB} is the size of on-chip memory (Source buffer, destination Buffer). Since the size of on-chip memory is limited, we need to partition the graph such that each partition fits the on-chip memory, for example, $N_{SB} = k \times b$. Using a smaller feature partition size b, vertex partition size k can be larger, and then there will be more data reuse within an edge block. However, because edges need to be loaded f/b times. Using a smaller b, loading edges will cause massive memory traffic. For example, if $b = 1$, edges need to be loaded for f times, leading to extensive memory traffic. Using a larger b, the data reuse within an edge block is limited. Besides, we consider the architectural parameters of external memory. The data transmission in DDR4 is organized in burst mode. Each *read/write* operation transmits consecutive 64 Bytes data. If features use 32-bit floating-point format. Each *read/write* operation transmits consecutive 16 features. Suppose $b < 16$, $\frac{16-b}{16}$ memory bandwidth will be wasted for loading a feature slice of length b, leading to degraded performance. Through our analysis as well as experiments (Section VI-A), the burst length $b = 16$ is selected as the optimal partition size.

Load imbalance: To resolve load imbalance, we leverage a centralized load balancing scheme [35] to allocate the tasks for FAMs. We maintain a task pool containing all 3-D partitions. When a FAM completes the feature aggregation of a 3-D partition, it is assigned another 3-D partition from the task pool. This resolves the load imbalance caused by the uneven degree distribution.

C. Feature Update Module

Feature Update Modules perform the feature transformation $(\tilde{\boldsymbol{A}}\boldsymbol{X})\boldsymbol{W}$ and element-wise activation. In GCN, weight matrix $\boldsymbol{W}$ has high density which is near 100%. However, the density of feature matrix $\boldsymbol{X}^l, (0 \leqslant l \leqslant L)$ can range from 1% to 100% depending on (1) property of datasets, (2) activation function. A single architecture is unable to achieve high efficiency under different densities. GCN framework needs to adapt to the variability. Previous works [16], [10] do not consider this variability, leading to inefficiency within feature transformation when the sparsity of feature matrix is high. Therefore, BoostGCN provides two types of FUM modules to handle different situations, (1) Sparse Feature Update Module (Sparse-FUM) (2) Dense Feature Update Module (Dense-FUM). Their architectures are shown in Figure 7.

Sparse-FUM: Sparse-FUM is deployed when feature matrices have low density (density of $\boldsymbol{X} < 10\%$). The architecture of Sparse-FUM is similar to FAM. Each iteration, sparse-FUM processes a batch of feature vectors produced by FAM. For example, z_1^{l-1}, z_2^{l-1}, z_3^{l-1} are concatenated to a matrix. The Format Transformation Module transforms the matrix to COO format by attaching the src index and dst index to the non-zero elements. In GCN, weight matrix is small, which can be stored on-chip in the Weight Buffer. The computation process of Sparse-FUM is similar to FAM. After feature update, Sparse-FUM produces h_1^{l-1}, h_2^{l-1}, h_3^{l-1} in dense format, becoming the input for the next layer.

Dense-FUM: If the feature matrices have high density, Sparse-FUM becomes inefficient. Then Dense-FUM is used for feature update. In Dense-FUM, we implement a 2-D systolic array. The systolic array performs the multiplication between a batch of feature vectors and weight matrix, using the output stationary dataflow [36]. If the size of systolic array is $m_1 \times m_2$, the total computation parallelism of c Dense-FUMs is $c \times m_1 \times m_2$.

Density Threshold: BoostGCN selects between Sparse-FUM and Dense-FUM based on the density of $\boldsymbol{X}$. The goal of setting the density threshold is to maximize the computation

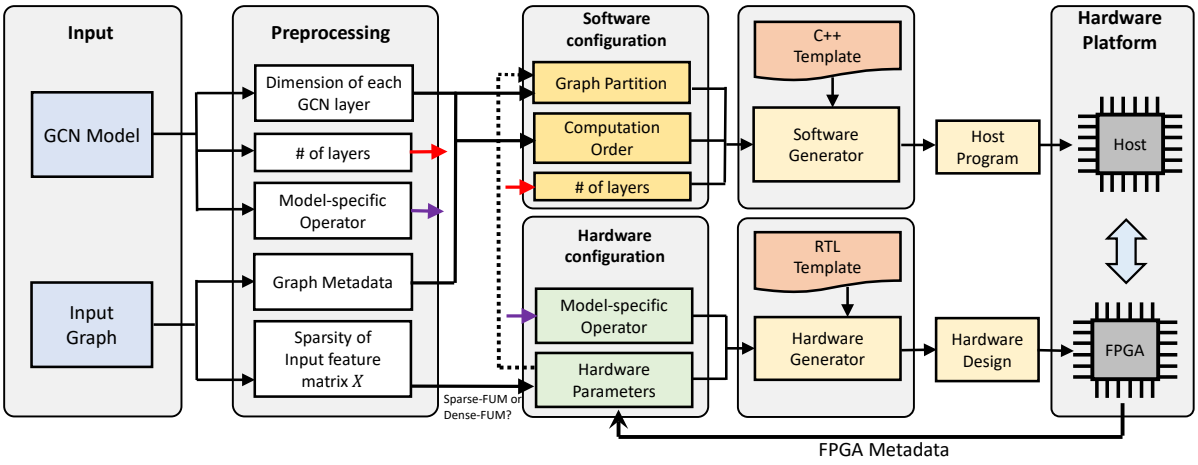

Fig. 8. Framework overview.

parallelism while simultaneously considering computation efficiency. On the one hand, to achieve the same parallelism, Sparse-FUM will consume more hardware resources than Dense-FUM. On the other hand, Dense-FUM has low efficiency when data is very sparse. Moreover, some extra overheads of Sparse-FUM need to be taken into consideration, such as (1) the overhead of pipeline stalls due to the bank conflicts in Weight Buffer and Result Buffer, and (2) the overhead of Format Transformation Module. Suppose each pipeline in Sparse-FUM consumes N_S resources (e.g. DSPs) and each MAC in Dense-FUM consumes N_D resources, the density threshold T_{th} is estimated by $\alpha * \frac{N_D}{N_S} (0 < \alpha < 1)$, where α is a factor of other overheads in Sparse-FUM determined by estimation. The Sparse-FUM is used when the density of X is smaller than T_{th}.

D. Task Scheduling Optimization

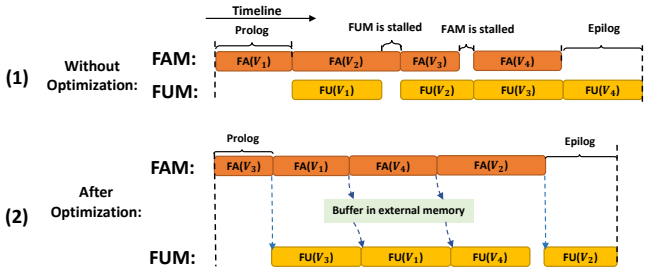

Fig. 9. An example of our proposed task scheduling optimization within the GCN-layer operation. FA(V_i) and FU(V_i) denote the feature aggregation and the feature update of interval V_i, respectively.

Previous works [10], [16] exploit the task-level pipelining strategy as shown in Figure 9-(1). The aggregated feature vectors produced by FAM are sent to FUM for feature update. However, their pipelining strategy can potentially lead to inefficiency due to the pipeline stall. The root cause is the uneven distribution of vertex degree among intervals. For example, if an interval has large average vertex degree, the feature aggregation of this interval will have long execution time, which potentially leads to the stall of FUM. If an interval has small average vertex degree, the feature aggregation has short execution time, which potentially leads to stalling of FAM.

In order to reduce the pipeline stalls, we propose a task scheduling optimization as illustrated in Figure 9-(2). First, we sort the intervals by their vertex degrees. The interval with small vertex degree will be executed first. Second, we allocate a buffer in external memory to store the aggregated feature vectors produced by FAM in case that FUM is not ready to consume new aggregated feature vectors. FUM can load the aggregated feature vectors from external memory afterward. Under the above strategy, we reduce (1) the pipeline stall of FUM by the intervals with large vertex degree, and (2) the pipeline stall of FAM by the intervals with small vertex degree. Notice that the sorting has the complexity of $O(\log(\frac{N}{k}))$, and the memory traffic of *store/load* the aggregated feature vectors is negligible compared with the memory traffic of feature aggregation. Thus, the overhead of our task scheduling is negligible.

E. Computation Order Optimization

GCN layer operation $\boldsymbol{AXW}$ has associativity that changing the computation order from aggregation first order $(\boldsymbol{AX})\boldsymbol{W}$ to update first order $\boldsymbol{A}(\boldsymbol{XW})$ will not affect the final result. choosing proper computation order, we can reduce the number of floating-point operations and external memory traffic based on the *hidden dimension/feature length* f of GCN layers. For instance, using Cora dataset, we can perform update first to reduce the feature length from 1433 to 16/128. Then the smaller feature length leads to less memory traffic and reduced number of floating point operations in aggregation phase. We specify two computation orders (1) In **aggregation first order**, we have the execution order: FA(l) $\rightarrow$ FU(l), where FA(l), FU(l) denote the feature aggregation, feature update of l^{th} layer respectively. In this order, we perform pipelined execution between feature aggregation of l^{th} layer and feature

update of l^{th} layer. (2) In **update first order**, we have the execution order: FA$(l-1) \rightarrow$ FU(l). We interleave the GCN operation of $(l-1)^{th}$ layer and l^{th} layer in a pipeline fashion. We perform the feature aggregation of $(l-1)^{th}$ layer first and then perform the feature update of the l^{th} layer. FA$(l-1)$, FU(l) are executed on hardware modules in pipelined fashion.

V. Framework Overview

Figure 8 depicts the framework overview of BoostGCN. The inputs to BoostGCN are the GCN model and the input Graph. BoostGCN extracts information from the input graph and the GCN model, and generates software configuration and hardware configuration. At the hardware end, BoostGCN decides the type of FUM to use based on the sparsity of the input feature matrix (Section IV-C). Once the hardware modules are decided, BoostGCN generates the hardware parameters for each hardware module based on the available hardware resources (FPGA Metadata, such as DSPs, ALMs, etc.). The hardware generator produces hardware design based on the RTL (Register-Transfer Level) templates. The hardware design will be deployed on the FPGA board. At the software end, BoostGCN performs graph partitioning by choosing (k, b) based on on-chip memory size. BoostGCN decides the computation order based on the graph metadata $(|\mathcal{V}|, |\mathcal{E}|, f_0)$ and the dimension of each GCN layer $(f_i, 1 \leqslant i \leqslant L)$. It uses the same partition size (k, b) and the same computation order for each GCN layer. Also, it generates the host program using the developed C++ template. The host program will be deployed on the host processor.

VI. Experiments

We use BoostGCN to generate the hardware designs and use the Stratix 10 GX 10M as the target platform for implementation. The external memory has 77 GB/s peak memory bandwidth with four memory channels. Each channel can provide 19.2 GB/s peak memory bandwidth. We synthesize our design and perform place-and-route using Quartus Prime Pro 20.2. The hardware resource utilization and frequency are obtained from the generated report by place-and-route. We compare our performance with state-of-the-art GCN frameworks PyG (Pytorch Geometric [8]) and DGL (Deep Graph Library [9]). PyG and DGL run on high-end CPU and GPU platforms. The details are shown in Table IV. We also compare our work with the state-of-the-art GCN accelerators–ASAP2020 [10] and HyGCN [16]. Following the convention in [16] and [15], we measure the execution time of two-layer Vanilla-GCN model. The GCN layer parameters are shown in Table III, which are summarized from [6], [16], [15], [10]. The reported execution time is the end-to-end latency, including the time of loading the input data from external memory and writing the results to external memory.

TABLE V
Design configurations used in experiments and resource utilization

Dataset	Configuration	Resource Utilization
Cora, Citeseer, Pubmed, PPI	**Data Partition Size**: $k = 4096, b = 16$ **FAMs**: $(p, b_{\text{FA}}, q = 8, 16, 4)$	$359K$ ALMs 3584 DSP blocks 4 MB
	Data Partition Size: $k = 512, b = 16$ **Sparse-FUMs**: $(p, b_{\text{FU}}, q = 8, 16, 4)$	
Flickr, Reddit, Yelp, Amazon	**Data Partition Size**: $k = 16384, b = 16$ **FAMs**: $(p, b_{\text{FA}}, q = 8, 16, 4)$	$294K$ ALMs 3840 DSP blocks 18 MB
	Dense-FUMs: $(c, m_1, m_2 = 8, 16, 16)$.	

Design Configurations: For FAM and Sparse-FUM, we use four pipelines in each processing element (PE) because FAM can read 4 edges from external memory in each clock cycle. For the four-pipeline PE, $\frac{N_D}{N_S}$ is equal to 0.28 for Sparse-FUM (Section IV-C) using the Intel floating-point intellectual property (IP) blocks (e.g. Multiplier, Accumulator). Then, we empirically estimate $T_{\text{th}} = 10\%$ as the threshold considering other overheads in Sparse-FUM. For Cora, Citeseer, Pubmed and PPI in Table II, we use the Sparse-FUM because the density of the input feature matrices are less than or close to T_{th}. For Flickr, Reddit, Yelp, and Amazon, we use Dense-FUM, because their input feature matrices have high density ($\approx 50\%$). Since the DDR4 memory we used has fixed data transaction burst length equal to 16, we set data partition size b as well as the hardware configurations $b_{\text{FA}}, b_{\text{FU}}$ to 16. The same feature partition size $b = 16$ is used for all GCN layers. The vertex partition size k is set based on the size of the on-chip buffer, such Source Buffer (Figure 4). The configurations of different designs with the corresponding FPGA resource utilization are reported in Table V.

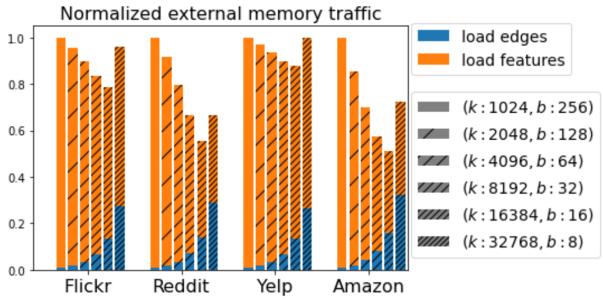

Fig. 10. Normalized external memory traffic for various partition sizes for $f_1 = 256$.

A. Performance of External Memory

External memory traffic: We profile the DRAM performance using Ramulator [37]. We fix the capacity of Source buffer in FAM and change partition size (k, b). Then, we evaluate the external memory traffic by measuring the number of DRAM read/write transactions under different partition sizes. Note that evaluation results for Cora, Citesseer, Pubmed, and PPI are not included. Since their input feature matrices can be fully stored in the on-chip memory. The results for Flickr, Reddit, Yelp, and Amazon are shown in Figure 10. Compared with the partition size $(k = 1024, b = 256)$,

TABLE IV
RESOURCES USED BY VARIOUS DESIGNS

	PyG-CPU / DGL-CPU	PyG-GPU /DGL-GPU	HyGCN [16]	ASAP2020 [10]	This work with Dense-FUM
Computation Resources	Intel Xeon Gold 5120 CPU @ 2.20 GHz 28 cores with 56 threads	Titian Xp, 1405 MHz 3840 CUDA cores	1 GHz @ 32 SIMD 16 cores and 8 systolic modules (each with 4 x 128 array)	128/256 accumulators 24 x 24 systolic array 250 MHz	$294K$ ALMs 3840 DSP blocks 250 MHz
Peak Performance	0.98 TFLOPS	9.3 TFLOPS	4.6 TFLOPS	0.21 TFLOPS	0.89 TFLOPS
On-chip Memory	L1d cache: 32 KB L1i cache 32 KB L2 cache: 1024 KB L3 cache: 19712 KB	L1 cache: 48 KB (per SM) L2 cache, 3 MB	128 KB (Input), 2 MB (Edge), 2 MB (Weight), 4 MB (Output) and 16 MB (Aggregation)	N/A	18 MB
External Memory Bandwidth	230 GB/s DDR4	547 GB/s	256 GB/s HBM 1.0	77 GB/s	77 GB/s DDR4

partition size $(k = 16384, b = 16)$ reduces external memory traffic of Flickr, Reddit, Yelp, Amazon by 21.6%, 44.6%, 12%, 49.1% respectively due to the increased data reuse under larger vertex partition size k. When the partition size becomes $(k = 32768, b = 8)$, we need to load edges 2 times compared with that of $(k = 16384, b = 16)$. Moreover, as discussed in Section IV-B, when $b = 8$, some data transactions are wasted by half, which could potentially lead to larger number of data transactions of loading features. So, 3-D partitioning based on data transaction burst length $b = 16$ achieves the best performance for PCFA.

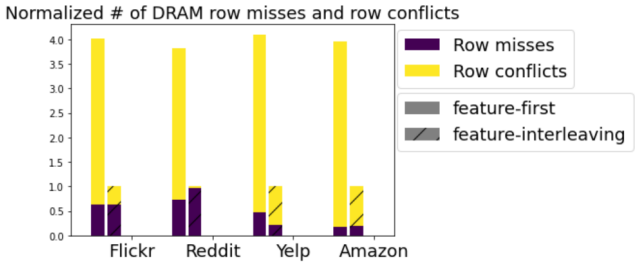

Fig. 11. Normalized number of DRAM row misses and row conflicts

DRAM latency: DRAM row misses&conflicts are the major factors causing large DRAM access latency. Memory accesses with large number of DRAM row misses&conflicts can potentially lead to hardware under-utilization, thus degraded performance. We measure the averaged number of DRAM row misses&conflicts under the feature-first order and our proposed feature-interleaving order. As shown in Figure 11, our proposed feature-interleaving order reduces the total number of row misses&conflicts by nearly 75% for each dataset, thus reducing the overall DRAM access latency.

B. Evaluation of the proposed optimizations

TABLE VII
AVERAGE SPEEDUP DUE TO OUR TASK SCHEDULING OPTIMIZATION $(f_1 = 128/256)$

	$k = 16384$ $b = 16$	$k = 8192$ $b = 32$	$k = 4096$ $b = 64$	$k = 2048$ $b = 128$	$k = 1024$ $b = 256$
Yelp	1.01×	1.03×	1.08×	1.1×	1.1×
Reddit	1.21×	1.22×	1.23×	1.24×	1.33×
Flickr	1.14×	1.10×	1.07×	1.05×	1.05×
Amazon	1.02×	1.03×	1.03×	1.04×	1.05×

Task scheduling optimizations: As discussed in Section IV-D, intra-stage load imbalance leads to stalled execution of FUM. This phenomenon is evident in real-world graphs due to the power-law degree distribution. We evaluate our Task scheduling optimizations through experiments (Table VII). The baseline is the task scheduling strategy used by [10], [16]. On average, our proposed optimizations achieve $1.06\times$, $1.24\times$, $1.08\times$, $1.04\times$ speedup on Yelp, Reddit, Flickr, Amazon, respectively.

Computation order: We compare the execution time of four datasets–Flickr, Reddit, Yelp, Amazon under various computation orders (Table IX). For Flickr, Reddit, and Amazon, the execution time using update-first order is less than the execution time under aggregation-first order. This is attributed to the fewer floating-point operations and less external memory traffic by using the update-first order. For example, the GCN layer parameter of Flickr is $f_0 = 500, f_1 = 128/256, f_2 = 7$. By transforming the feature length from 500 to $128/256$ using the update-first order, we reduce the number of floating operations and the memory traffic of feature aggregation.

TABLE IX
COMPARISON OF EXECUTION TIME USING VARIOUS COMPUTATION ORDERS $(f_1 = 128/256)$

	Update-First	Aggregation-First
Flickr	20.1 ms / 40.4 ms	21.2 ms / 43.6 ms
Reddit	98.1 ms / 188.1 ms	192.8 ms / 227.3 ms
Yelp	193.0 ms / 332.5 ms	218.1 ms / 281.8 ms
Amazon	793.5 ms / 1277.0 ms	961.3 ms / 1374.0 ms

C. Comparison with the state-of-the-art

We compare our work with the advanced and well-optimized frameworks PyG [8], DGL [9], ASAP2020 [10]. First, we compare the execution time on Cora, Citeseer, Pubmed. As shown in Table VIII, our work achieves significant speedup compared with PyG and DGL. Compared with prior FPGA framework ASAP2020 [10], we achieve $45\times$ speedup on average. Note that ASAP2020 [10] does not exploit data sparsity in the update phase. Our proposed sparse-FUM can efficiently skip the zero elements in the vertex features. We also evaluate our performance on other five datasets. Our implementation significantly outperforms PyG and DGL. On datasets Flickr, Reddit, Yelp, our implementation achieves $3\times$ speedup on average than ASAP2020 [10] as shown in Table

TABLE VI
COMPARISON OF EXECUTION TIME ON PPI, FLICKR, REDDIT, YELP, AMAZON ($f_1 = 128/256$)

Dataset	PyG-CPU	PyG-GPU	DGL-CPU	DGL-GPU	ASAP2020 [10]	HyGCN [16]	This work
PPI	34.6 ms / 190 ms	34.4 ms / 71.9 ms	80 ms / 88 ms	15 ms / 15 ms	N/A	N/A	**1.5 ms** / **2.8 ms**
Flickr	3.3 s / 6.69 s	337 ms / 357 ms	300 ms / 680 ms	26 ms / **34 ms**	48.7 ms / 101.7 ms	N/A	**20.1 ms** / 40.4 ms
Reddit	81 s / N/A ⋆	OoM‡	3.2 s / 8 s	390 ms / 650 ms	598.7 ms / 703 ms	289 ms / N/A	**98.1 ms** / **188.5 ms**
Yelp	56 s / N/A	OoM	3.5 s / 5.2 s	350 ms / 470 ms	306.3 ms / 556.0 ms	N/A	**193.0 ms** / **281.8 ms**
Amazon	N/A	OoM	36 s / 45 s	OoM	N/A	N/A	**793.5 ms** / **1277.0 ms**

‡ OoM: Out of memory. ⋆ N/A: Data not available.

TABLE VIII
COMPARISON OF EXECUTION TIME ON CORA, CITESEER, AND PUBMED ($f_1 = 16/128$)

	Cora	Citeseer	Pubmed	Comments
PyG-CPU	2.5 ms / 17.1ms	3.8ms / 22ms	15 ms / 229 ms	
PyG-GPU	637 us / 945 us	805 us / 1.5 ms	1.1 ms / 3.4 ms	
DGL-CPU	2.2 ms / 12.7 ms	8.2 ms / 17 ms	9.7 ms / 22 ms	
DGL-GPU	0.98 ms / 1.1 ms	1.1 ms / 1.2 ms	1.2 ms / 1.3 ms	
ASAP2020 [10]	687 us / 3.5 ms	2.1 ms / 11.1 ms	2.3 ms / 9.5 ms	
HyGCN [16]	N/A / **21 us**	N/A / 300 us	N/A / **640 us**	ASIC design at 1 GHz
This work	**19.4 us** / 76.5 us	**25.3 us** / **125.8 us**	**166.2** us / 1140 us	FPGA design at 250 MHz

VI. The improvement is due to (1) less memory traffic (2) less DRAM row misses&conflicts, (3) our proposed scheduling optimization that reduces the overall pipeline stall.

HyGCN [16] is based on advanced ASIC technology. It uses 4096 32-bit fixed-point multipliers and 512 32-bit fixed-point ALUs running at 1 GHz. In contrast, our work uses 3584 32-bit floating-point multipliers at 250 MHz. HyGCN architecture has peak performance of 4.6 TFLOPS while the design used in this work has peak performance of 0.89 TFLOPS. HyGCN exploits High Bandwidth Memory (HBM) with peak memory bandwidth of 256 GB/s while peak memory bandwidth of our platform is 77 GB/s. Despite the vast gap in the peak performance and memory bandwidth, we still achieve less execution time on Reddit and Citeseer. The improvement is due to (1) the PCFA scheme that greatly reduces the memory traffic and overall memory access latency, and (2) Sparse-FUM that efficiently skips zero elements within feature update.

VII. CONCLUSION

In this paper, we proposed the BoostGCN framework to optimize GCN inference with improved memory performance and massive data parallelism. Our proposed PCFA is an algorithm-architecture co-optimization scheme that significantly improves the efficiency of feature aggregation. The proposed hardware architecture facilitates pipelined execution of the two computation phases. The accelerators generated by our framework achieve $100\times$, $30\times$, $(3\text{-}45)\times$ speedup over CPU, GPU, and prior FPGA accelerators. In the future, we plan to develop a design space exploration algorithm that can fully automate the design process.

ACKNOWLEDGMENT

This work was supported by the U.S. National Science Foundation (NSF) under grants OAC-1911229 and CCF-1919289, and in part by Intel.

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] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D.-Y. Yeung, "Gaan: Gated attention networks for learning on large and spatiotemporal graphs," *arXiv preprint arXiv:1803.07294*, 2018.

[4] B. Yu, H. Yin, and Z. Zhu, "Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting," in *IJCAI*, 2018.

[5] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting," in *International Conference on Learning Representations*, 2018.

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

[7] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur, "Protein interface prediction using graph convolutional networks," in *Advances in Neural Information Processing Systems*, 2017, pp. 6530–6539.

[8] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[9] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," *arXiv preprint arXiv:1909.01315*, 2019.

[10] 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.

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

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

[13] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graphsaint: Graph sampling based inductive learning method," in *International Conference on Learning Representations*, 2019.

[14] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2018.

[15] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt *et al.*, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 922–936.

[16] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 15–29.

[17] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 45–54.

[18] R. Rajat, H. Zeng, and V. Prasanna, "A flexible design automation tool for accelerating quantized spectral cnns," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 144–150.

[19] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[20] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2018.

[21] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "Hybriddnn: A framework for high-performance hybrid dnn accelerator design and implementation," *arXiv preprint arXiv:2004.03804*, 2020.

[22] Y. Meng, S. Kuppannagari, R. Kannan, and V. Prasanna, "Dynamap: Dynamic algorithm mapping framework for low latency cnn inference," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 183–193.

[23] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[24] D. Kim, J. Ahn, and S. Yoo, "A novel zero weight/activation-aware hardware architecture of convolutional neural network," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1462–1467.

[25] Y. Niu, R. Kannan, A. Srivastava, and V. Prasanna, "Reuse kernels or activations? a flexible dataflow for low-latency spectral cnn acceleration," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 266–276.

[26] 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.

[27] 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.

[28] G. Dai, Y. Chi, Y. Wang, and H. Yang, "Fpgp: Graph processing framework on fpga a case study of breadth-first search," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 105–110.

[29] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 217–226.

[30] Z. Shao, R. Li, D. Hu, X. Liao, and H. Jin, "Improving performance of graph processing on fpga-dram platform by two-level vertex caching," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 320–329.

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

[32] 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.

[33] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, "Gnnadvisor: An efficient runtime system for gnn acceleration on gpus," *arXiv preprint arXiv:2006.06608*, 2020.

[34] C. Tian, L. Ma, Z. Yang, and Y. Dai, "Pcgcn: Partition-centric processing for accelerating graph convolutional network," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 936–945.

[35] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *Introduction to parallel computing*. Pearson Education, 2003.

[36] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.

[37] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2015.