FPGA Acceleration of GCN in Light of the Symmetry of Graph Adjacency Matrix

Gopikrishnan Raveendran Nair*[1], Han-Sok Suh*[1], Mahantesh Halappanavar[3], Frank Liu[2], Jae-sun Seo[1], and Yu Cao[1]

[1]School of ECEE, Arizona State University, Tempe, AZ, USA
[2]Oak Ridge National Lab, Oak Ridge, TN, USA
[3]Pacific Northwest National Labs, Richland, WA, USA

***Abstract*—Graph Convolutional Neural Networks (GCNs) are widely used to process large-scale graph data. Different from deep neural networks (DNNs), GCNs are sparse, irregular, and unstructured, posing unique challenges to hardware acceleration with regular processing elements (PEs). In particular, the adjacency matrix of a GCN is extremely sparse, leading to frequent but irregular memory access, low spatial/temporal data locality and poor data reuse. Furthermore, a realistic graph usually consists of unstructured data (e.g., unbalanced distributions), creating significantly different processing times and imbalanced workload for each node in GCN acceleration.**

To overcome these challenges, we propose an end-to-end hardware-software co-design to accelerate GCNs on resource-constrained FPGAs with the features including: (1) A custom dataflow that leverages symmetry along the diagonal of the adjacency matrix to accelerate feature aggregation for undirected graphs. We utilize either the upper or the lower triangular matrix of the adjacency matrix to perform aggregation in GCN to improve data reuse. (2) Unified compute cores for both aggregation and transform phases, with full support to the symmetry-based dataflow. These cores can be dynamically reconfigured to the systolic mode for transformation or as individual accumulators for aggregation in GCN processing. (3) Preprocessing of the graph in software to rearrange the edges and features to match the custom dataflow. This step improves the regularity in memory access and data reuse in the aggregation phase. Moreover, we quantize the GCN precision from FP32 to INT8 to reduce the memory footprint without losing the inference accuracy. We implement our accelerator design in Intel Stratix10 MX FPGA board with HBM2, and demonstrate 1.3×-110.5× improvement in end-to-end GCN latency as compared to the state-of-the-art FPGA implementations, on the graph datasets of Cora, Pubmed, Citeseer and Reddit.

I. Introduction

Graph Convolutional Neural Networks (GCNs) are the state-of-the-art machine learning method to process the graphs. GCNs take a high dimensional representation of graph data and transform it into lower dimension representation, without altering the graph structure [1], [2]. In general, GCNs consist of two main operations: aggregation and transformation, as shown in Figure 1. In feature aggregation, each node takes the feature vector associated with its neighboring nodes and aggregates them to produce a new feature vector. The feature transformation phase transforms the feature matrix of the nodes to a new dimension, using a weight matrix. This operation is similar to that in a multi-layer perceptron (MLP).

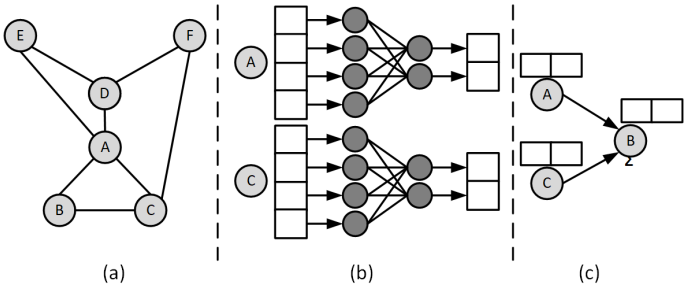

Fig. 1. Data processing in GCN: (a) Input graph, (b) Transformation of two nodes, and (c) Aggregation of a node.

Acceleration of GCNs on hardware platforms is challenging due to its sparsity, randomness, and non-uniformity. The adjacency matrix of a GCN is a sparse data structure leading to irregular memory accesses, low spatial/temporal data locality and reuse [2]. Consequently, execution time and memory access patterns in aggregation are unpredictable and irregular [3]. On the other hand, feature transformation is a dense matrix multiplication with regular memory access and high data reuse. Therefore, GCN accelerators should be capable of handling both the sparsity and irregularity of aggregation, and the dense and regular computation in feature transformation [3].

As feature aggregation is unique to GCNs and presents a major challenge, we propose a *custom execution dataflow to efficiently accelerate feature aggregation* in this work. By exploiting the symmetry of adjacency matrices in undirected graphs, we utilize either the upper or the lower triangular matrix to perform aggregation in GCN. Furthermore, we develop a dynamically reconfigurable compute core to unify the processing for aggregation and transformation. This approach minimizes the balancing issues in workload and pipelining in GCN operations.

The major contributions of this work are as follows:

- We propose a dataflow to accelerate feature aggregation by exploiting the symmetry of the adjacency matrix. Such a dataflow reduces redundant data movement and improves data reuse.
- We design a dynamically reconfigurable compute core and associated control logic to support the heterogeneous workload across aggregation and transformation, as well

* These authors contributed equally and are co-first authors.

This work is partially supported by C-BRIC, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. It is also supported in part by the U.S. Department of Energy, through the Office of Advanced Scientific Computing Research's "Data-Driven Decision Control for Complex Systems (DnC2S)" project.

978-3-9819263-7-8/DATE23/© 2023 EDAA

as the symmetry nature of feature aggregation.

- We design a preprocessing method to rearrange the edges and features of the graph in order to match the data flow. This step ensures more regular memory access and better data reuse in the aggregation phase.
- We implement our design on Intel Stratix 10 MX FPGA board with HBM2, and achieve significant improvement in end-to-end latency: 1,101× and 202× over PyG [4] and DGL [5] on CPUs, 13× and 17× over PyG and DGL on GPUs, and 1.3×-110.5× over prior FPGA accelerators [6], [7] on four graph datasets.

II. Background and Motivation

A. GCN Operations

GCNs have a neighborhood aggregation scheme where the features of neighbouring nodes are recursively transformed and aggregated to generate a new representation for the target nodes, which is commonly deployed in GCN [1], GraphSage [8], and GAT [9]. In this work, we focus on the inference of one the most popular model, GCN in [1].

Primary operations in the GCN include transformation and aggregation, as shown in Fig. 1. The output of a layer ($\mathbf{X}^l$) can be represented as: $X^l = \text{ReLU}(\tilde{A}X^{(l-1)}W^l)$. $\tilde{A}$ is the normalized adjacency matrix, given by $\tilde{A} = D^{-1/2}AD^{-1/2}$, where D is the degree matrix of the graph. Here the transformation operation can be represented by $X^{l'} = X^{(l-1)}W^l$, and the aggregation operation is represented by $X^l = \tilde{A}X^{(l')}$.

In the transformation phase, the features ($\mathbf{X}$) of the graph nodes are multiplied with a weight matrix ($\mathbf{W}$) to transform the features. During aggregation, each node's feature is aggregated with the feature of its neighboring nodes to generate a new representation. The output is then passed through an activation function to produce the output of a layer in GCN. These operations are repeated for all layers within a GCN.

B. Hardware Acceleration of GCNs

Recent works demonstrate effective hardware acceleration of GCNs over CPUs and GPUs [3], [6], [7]. A GCN accelerator should be able to manage both the dense transformation phase and the sparse aggregation phase. To that end, design challenges remain open on the following aspects:

1) Imbalanced workload: Existing works use separated compute engines for transformation or aggregation [3], [6], [7]. Since the latency of these two stages are imbalanced, such a design usually encounters pipeline stalls [6]. In [6], when pipeline stalls occur, the generated output of a compute engine has to be written back to external memory, at the cost of more external memory accesses. To overcome this, we unify the compute core, which can be dynamically configured into an aggregation or transformation engine based on the computation flow.

2) Data reuse under the symmetry: The randomness of aggregation makes it difficult to ensure data reuse. Distinguished from [3], [6], our hardware architecture exploits the symmetry in the adjacency matrix to maximize data reuse: Given an undirected graph, the upper half and the lower half of its adjacency matrix contain the same information and thus, in principle, we only need one half of the adjacency matrix to complete the aggregation phase. We further address this in data pre-processing on software to match the custom dataflow.

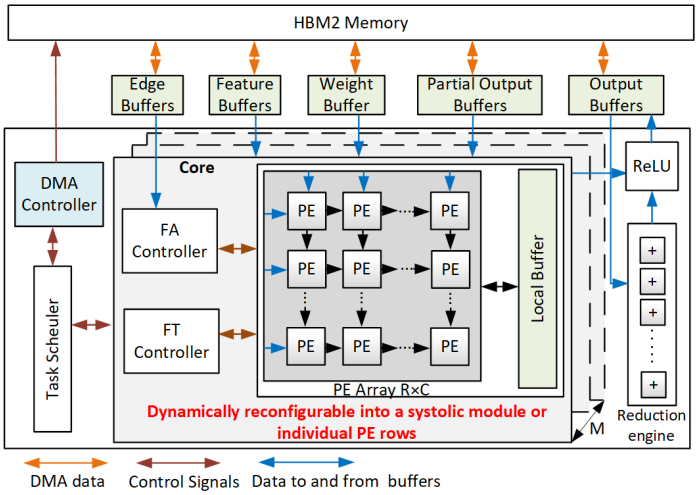

Fig. 2. The dynamically reconfigurable FPGA accelerator for GCNs.

3) GCN quantization: Previous works suffer from high memory footprint due to large graph size and 32-bit floating point data representation [3], [6], [7]. To that end, we adopt quantization in GCN training to reduce data representation from 32-bit floating point to 8-bit fixed point. This solution helps reduce memory requirement on FPGA, without sacrificing the inference accuracy.

III. FPGA Architecture and Dataflow

A. Architecture Overview

Fig. 2 shows the architecture of the GCN accelerator. It consists of six compute cores, a task scheduler, a reduction engine, the DMA controller, off-chip HBM2 memory, and on-chip buffers.

Each core in the accelerator supports the unified operation for aggregation and transformation. It consists of a scheduler for aggregation or transformation, an array of processing elements (PEs), and a local buffer. The aggregation and transformation scheduler generates corresponding control signals to the core. The local buffer stores the partial results of computation and feature data of nodes, which are reused by the PE array; it is exclusive for each core. The PE array is organized into R rows and C columns. They can work collectively as a systolic unit for transformation, or as individual rows of PEs for aggregation. Each PE is an 8-bit fixed point multiply-and-accumulate unit. For all cores inside the compute engine, the task scheduler generates control signals to enable either the aggregation scheduler or transformation scheduler in each core; the reduction engine accumulates the output produced across multiple cores.

We use multiple on-chip buffers for various purposes. The edge buffer contains the edge information of the graph in compressed co-ordinate format (COO). It is also used as an instruction buffer for the aggregation scheduler. Each core has its own edge buffer allowing parallel edge access. The

feature buffer stores the node features for aggregation and transformation. The weight buffer saves the weights for feature transformation. It is a banked memory with the number of banks equal to C in the PE array. The partial output buffer stores the partial output generated in the previous iteration to be consumed by the current iteration. The output buffer holds the output generated from the core for reduction engine. It is also a banked buffer with the number of banks equal to the number of compute cores (M). The data flow is designed to ensure that no inter-core communication is needed.

B. Symmetry of Adjacency Matrix

For an undirected graph, the upper and lower half of its adjacency matrix contain the same information, i.e., it is symmetric. Fig. 3(a) presents an example of the adjacency matrix of an undirected graph with six nodes. For a large graph, we need to partition the matrix into smaller tiles to fit into on-chip memory. As shown in Fig. 3(a), the six nodes are grouped into three vertex groups, V1, V2, and V3. The adjacency matrix is organized into 2×2 tiles, as (V1, V1), (V1, V2), etc.

In an undirected graph, the i^{th} row and i^{th} column of the adjacency matrix is the transpose of each other. This property is true for the tiles as well, e.g., tile (Vi, Vj) and tile (Vj, Vi) are the transpose of each other and contain the same information. Therefore, we only need to compute one of them and can skip the other, saving approximately half of the adjacency matrix in computing as highlighted in Fig. 3(a).

The aggregation operation is then performed tile by tile. For the tile (V1, V2), the features of the neighbors or source vertex group V2 are added to the destination vertex group V1 along the horizontal direction, as shown in Fig. 3(b). This produces a partial output (PO) denoted as POV1. Similarly, for the transpose pair (V2, V1), a partial output POV2 is produced. Due to the symmetry, we only need to compute one tile from a transpose pair. To compensate the horizontal aggregation PO from the discarded tile, we conduct an aggregation in the vertical direction in the selected transpose pair tile as shown in Fig. 3(c), i.e., the features of V2 is getting added to V1. For vertical aggregation, the source and destination vertices are interchanged as compared to the horizontal aggregation. Thus, except for the diagonal tiles, each tile produces two partial outputs, one from the vertical aggregation and the other from the horizontal aggregation. For the diagonal tiles there are no transpose pairs as the transpose of a diagonal tile is the same.

Since the adjacency matrix is extremely sparse, it is stored in the coordinate format (COO). This helps us skip zero entries inside the adjacency matrix. In the COO format, each edge is represented as (*row*, *col*, *value*). For horizontal aggregation, the *row* entry becomes the destination node and *col* entries as its neighbors, where for vertical aggregation the *col* entry becomes the destination and *row* entries as its neighbors. Thus, without any storage overhead, we can compute vertical and horizontal aggregation using the COO format.

In summary, by using the symmetry property for aggregation, we produce twice the number of outputs for non-diagonal tiles, as compared to the normal aggregation, ensuring more edge data reuse for the same set of edge inputs.

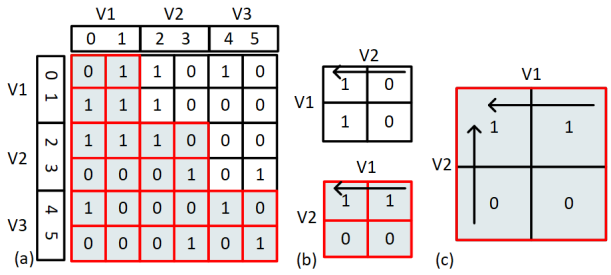

Fig. 3. Aggregation with tiles: (a) Adjacency matrix with compute tiles highlighted; (b) Symmetry on a pair of transpose tiles with the direction of aggregation; (c) Vertical and horizontal aggregation on a tile.

C. Custom Data Flow based on Tiles

Fig. 4 presents the dataflow of GCN acceleration based on the symmetry property. *FT input* is the input required for transformation and *FA input* for aggregation. Tile *i* represents the adjacency matrix of tile in the COO format and V*i* represents the features of the corresponding vertex group.

We perform feature transformation first followed by feature aggregation, to reduce the amount of computations [2]. For Iteration 1 in Fig. 4(b), the inputs are Tile 1 and features of V1. It produces a transformation output FTOV1. This output is then reused as the input for aggregation and produces one partial output POV1. Since it is a diagonal tile, only horizontal aggregation output is produced.

For Iteration 3 which is a non-diagonal tile, we have both vertical aggregation and horizontal aggregation. There is no transformation operation for Tile 3 as the feature inputs required for horizontal aggregation are the output of transformation of Tile 1 and feature the input for vertical aggregation is the transformation output of Tile 2. Thus, for a non-diagonal tile, we have only aggregation but no transformation. The aggregation of Tile 3 produces two sets of output POV2 and POV1. Here POV1 and POV2 is accumulated on top of the POV1 and POV2 from the previous iteration.

In Iteration 5, the aggregated output for vertex group V2 (OV2) is produced, and by the end of Iteration 6, the complete aggregated output for vertex group V1(OV1) and V3(OV3) is produced. As shown in Fig. 4(b), the order of computation is selected to maximize input reuse. By computing the diagonal tiles first and then the non-diagonal tiles in the same row, we ensure maximum data reuse of the output from the diagonal tiles. The process continues until all tiles are computed.

D. Mapping Dataflow to Hardware

1) Transformation: For transformation, the core is reconfigured into the systolic module. Each core has a PE array of R rows and C columns. Thus, each core can handle R rows of the feature matrix and C columns of the weight matrix. We have M such cores operating in parallel.

Fig. 5(a) shows the dataflow for transformation. Weights are fed from the top and is pushed to the systolic module, and features are fed from the left and pushed to the right in every cycle. To access data parallelly in every cycle, we have a feature matrix with banks set to R and weight matrix with banks set to C. For M parallel cores, we have M on-chip feature buffers. Since weights are the same within a GCN layer, we only need

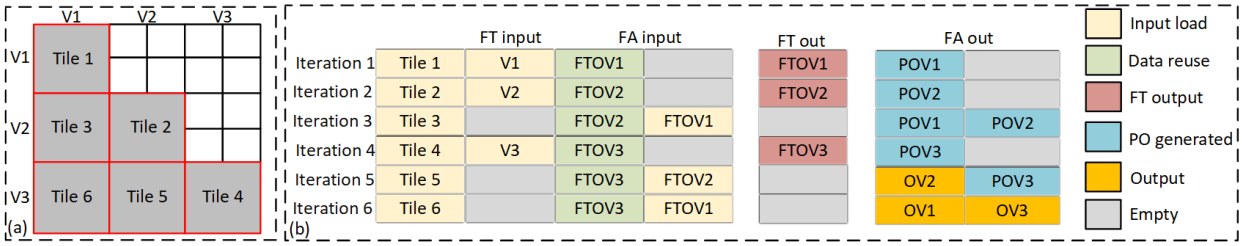

Fig. 4. Custom dataflow based on symmetric tiles: (a) Adjacency matrix with tiles numbered according to the order of execution; (b) Dataflow with input and output at each iteration step.

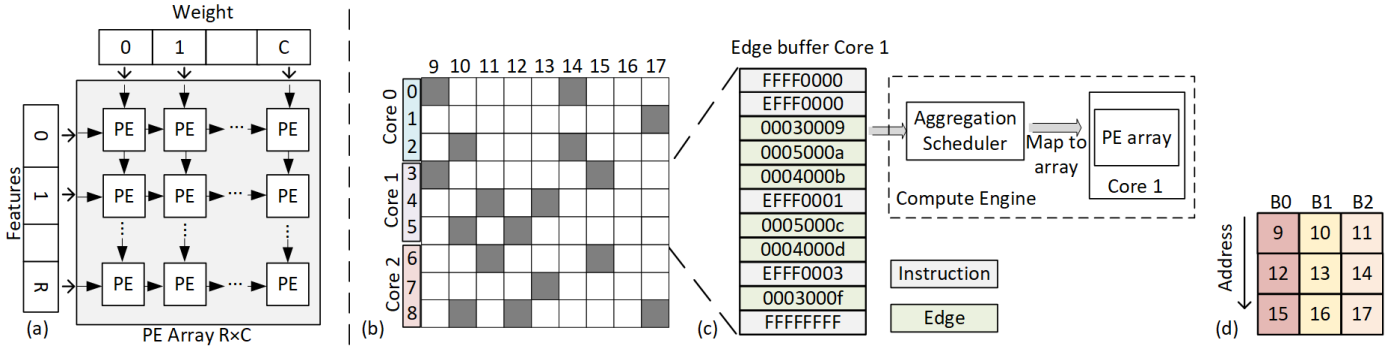

Fig. 5. Dataflow on FPGA: (a) Mapping of transformation into the PE array; (b) The adjacency matrix tile showing the interval partition; (c) Edge buffer for core 1 and mapping of edges from the buffer to the core; (d) Layout of node features inside a banked on-chip memory.

one weight buffer and the data is broadcasted to all cores. The FT scheduler generates the address to read data from the feature buffer and the weight address. Since all M cores produce output at the same time, we have the output buffer with M banks to enable parallel output writes. The peak parallelism we achieve here is M×R×C.

2) Aggregation: At the tile level, to improve the regularity in memory access and data reuse in the aggregation phase, we further partition each tile into smaller intervals. For example, if we have three cores, the destination nodes and its associated edges in each tile is equally split into three sets, as shown in Fig. 5(b). The edges in each set represented in the COO format is populated inside the edge buffer of each core.

For horizontal aggregation, the neighbors or source nodes of a tile are the same for all cores. Since all the cores compute in parallel, accessing the features of all the source nodes for aggregation will result in a memory bottleneck. Therefore we use a banked on-chip memory to store the source nodes. As the maximum number of parallel read requests equals to the number of cores, the number of banks for the on-chip memory is then set to the number of cores. Each consecutive source node feature is statically mapped into a unique bank as shown in Fig. 5(d). The edges fed into each core's edge buffer are arranged in the ascending order of source nodes. This increases the probability of multiple edges with the same source nodes or with the next source nodes to be computed one after the other. To exploit data reuse and locality in this data flow, whenever a core makes a read request to a specific address location, features of the source nodes at that address location across all banks are fetched and moved to that core's local buffer.

For vertical aggregation, the destination and source nodes are interchanged and each set of the source nodes is unique to a core. Therefore, a separate on-chip memory is set up for each core to store the features for vertical aggregation. In the case of vertical aggregation, each core produces an aggregated partial output for the same destination nodes, which needs to be combined to from the final output. We use a reduction engine which adds up the partial output from each core into a single output. Moreover, the reduction engine computes the output from the previous tile while the current tile is being executed, thereby hiding its latency. The edges are fed into the core sequentially. For each edge, it takes 4 clock cycles for the PE to compute. In each clock cycle, the aggregation scheduler fetches a new edge from the edge buffer and decodes the data to check whether it is a configuration instruction or an edge data. For each edge, the aggregation scheduler checks the PE status within the core for the availability of a free PE row. If all PE rows are busy, then aggregation of that corresponding core will be stalled until a PE row becomes available, or else the edge is assigned to a free PE row and its status is updated. To compute vertical and horizontal aggregation for non-diagonal tiles, the scheduler checks if two PE rows are free; if not, aggregation of that corresponding core will be stalled until two PE rows become available.

To improve the compatibility of input data to the custom flow, we further develop pre-processing codes using PyG [4]. The pre-processing codes accept the number of cores and tile size as input and generate the edges in order at the tile level following the custom dataflow. The generated output is then used to populate the main memory. The DMA module along with the top level scheduler will move the data tile by tile per the computation order.

IV. Results

A. Experimental Setup

We implement our design in Verilog and use the Intel Stratix10 MX board with HBM2 as our target platform. The

TABLE I
CHARACTERISTICS FOR FOUR DIFFERENT GCN DATASETS.

	Cora	Citeseer	Pubmed	Reddit
Nodes	278	3327	19717	232965
Edges	10556	9228	88651	114,615,892
$\mathbf{X}^0$	1433	3703	500	602
$\mathbf{X}^1$	128	128	128	128
$\mathbf{X}^2$	7	6	3	41
Layers	2	2	2	2

FPGA has 8Gb of HBM2 memory, 2,073k logic elements, 702k ALMs, 134Mb of M20K memory, 11Mb of MLAB memory and 3,960 DSPs. We use Intel Quartus 19.4 to map the design to FPGA. We use 8-bit fixed-point precision to represent weights and features, and 32-bit to represent each edge in COO format.

We fix our design parameters, including the number of rows of PE array in a core to 10 and the number of columns of PE array to 66. Thus a single core will use 660 DSPs and we can have 6 such cores totalling the DSP count to 3,960 on the FPGA board. Increasing the number of columns in a PE array will result in higher parallelism along the feature dimension; increasing the number of rows of a PE array will increase the number of nodes that can be computed in parallel. On the other hand, if the size of a column or a row is too big, it will negatively impact such parallelism. Therefore, in our design across the 6 cores, we select our total array as 60×66 to ensure a balanced parallelism. The tile size we use to partition the graph is set to 1,020. The impact of the tile size on the design are studied and reported in [6]. Similar to that, we select the smallest tile size to highlight the speedup from the symmetry-based acceleration. We measure the end-to-end latency of a two-layer GCN for the four widely used datasets: Cora, Citeseer, PubMed, and Reddit. The details of the datasets are shown in Table I.

B. Quantization of GCN Algorithms

To reduce the memory footprint, we experiment GCNs with lower precision. Fig. 6 presents the accuracy of the GCN for four different datasets with various bit precision values for both weights and features. We use the GCN structure in [1] and Deep Graph Library (DGL) [5] with PyTorch in this experiment. The accuracy drop is negligible for all datasets when we move from 32-bit floating point to 8-bit fixed-point precision. The difference between the baseline accuracy and accuracy at 8-bit is less than 3% for all four datasets. Beyond 8-bit, the accuracy rapidly drops for all datasets. Thus we adopt 8-bit quantization for weights and features since it provides comparable accuracy with 32-bit precision.

C. Speedup with the Symmetry Property

In this section, we compare the performance of two implementations, one with the symmetry property and the other without. Fig. 7 presents the number of tiles that need to be computed for the four datasets under the baseline case without the symmetry property and with the symmetry property. Our method requires up to 50% less compute tiles to complete end-to-end GCN computation. The reduction is more pronounced

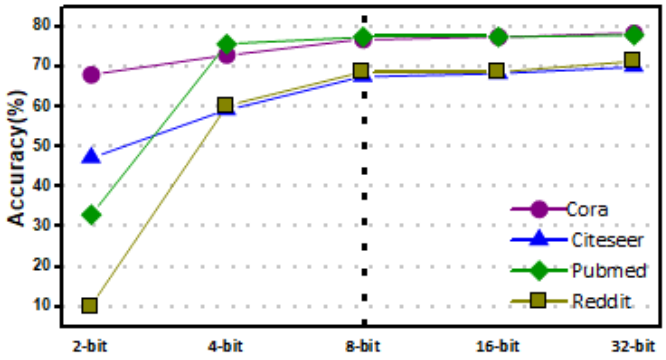

Fig. 6. Quantization of the GCN models. 8-bit is selected before experiencing any accuracy loss.

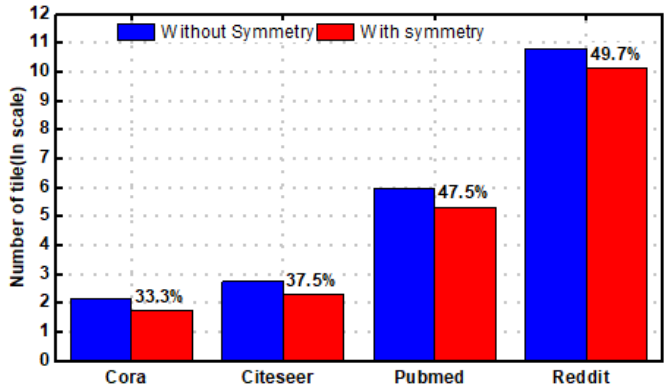

Fig. 7. The symmetry property effectively helps reduce the number of compute tiles in GCN acceleration.

for larger datasets. Furthermore, we set up a same implementation on FPGA, except the symmetry flow is not adopted. For Cora, the end-to-end latency without the symmetry property is 108.81µs. The adoption of the symmetry is 1.7× faster than that.

D. Comparison with State-of-the-Art Results

We compare our work with the state-of-the-art graph learning frameworks PyG [4] and DGL [5]. We also compare our design with the state-of-the-art FPGA implementations ASAP2020 [7] and BoostGCN [6]. We synthesize our design at 220MHz. The resource utilization on the FPGA board is summarized in Table II. Table III summarizes the comparison with the latest GCN accelerators, with Table IV evaluates the hardware resources across multiple design. Our resource is close to that in BoostGCN [6].

Compared to PyG and DGL, our work achieves up to 1,101× and 202× improvement in end-to-end latency in CPU and up to 13× and 17× in GPU implementations, respectively. Despite the vast difference in computing resources, our FPGA design

TABLE II
RESOURCE UTILIZATION OF FPGA FOR DIFFERENT DATASETS.

Dataset	ALMs	RAMs(MB)	DSPs
Cora	257208 (36.6%)	5.23 (28.86%)	3960 (100%)
Citeseer	243624 (34.67%)	6.12 (33.77%)	3960 (100%)
Pubmed	300288 (42.73%)	6.34 (34.98%)	3960 (100%)
Reddit	292441 (41.59%)	15.96 (88.06%)	3960 (100%)

TABLE III
COMPREHENSIVE EVALUATION OF EXECUTION TIME FOR DIFFERENT DATASETS ACROSS DIFFERENT IMPLEMENTATIONS.

Datset	PyG-CPU	PyG-GPU	DGL-CPU	DGL-GPU	HyGCN [3]	ASAP2020 [7]	BoostGCN [6]	Our Work	Over [6]
Cora	17.1ms	945µs	12.7ms	1.1ms	21µs	3.5ms	76.5µs	**62.77µs**	**1.20x**
Citeseer	22ms	1.5ms	17ms	1.2ms	300µs	11.1ms	125.8µs	**100.37µs**	**1.25x**
PubMed	229ms	3.4ms	22ms	1.3ms	640µs	9.5ms	1140µs	**882µs**	**1.28x**
Reddit	81s	out of mem	3.2s	390ms	289ms	598.7ms	98.1ms	**73.51ms**	**1.33x**

TABLE IV
HARDWARE RESOURCE USED BY VARIOUS DESIGNS.

	CPU	GPU	HyGCN [3]	ASAP2020 [7]	BoostGCN [6]	Ours
Compute Resources	Intel Xeon Gold5120 CPU 28 cores and 56 threads	Titan Xp 3840 cores	32 SIMD cores 8 systolic module	128/256 accumulators 24x24 systolic array	294k ALMs 3840 DSPs	**292k ALMs** **3960 DSPs**
Frequency	2.20GHz	1405MHz	1GHz	250MHz	250MHz	**220MHz**
Memory	20.8MB cache	48KB L1 cache 3MB L2 cache	24MB	N/A	18MB	**18MB**

achieves significant speedup over both the CPU and GPU implementations. These results confirm that CPUs and GPUs are more suitable for dense computations with regular memory access. On the other hand, the regular structure in CPUs and GPUs limits their capability in managing the heterogeneous and irregular nature of GCN computations.

We also achieve up to 110.5× improvement compared to ASAP2020 [7]. ASAP2020 uses a two-phase graph pre-processing, by removing the edge connections of high degree nodes and merging common neighbours together, and then use graph reordering algorithms to improve data locality.

We achieve up to 1.3× improvement over BoostGCN [6]. The speedup increases as the dataset size increases. This is because as the graph gets bigger, the reduction in the number of compute tiles from the symmetry property is more pronounced. BoostGCN employs sparse feature transformation which skips zero elements in the feature vectors during transformation. For Cora, Citeseer and Pubmed, it stores the entire input feature matrices on the memory. BoostGCN also partitions the graphs with a tile size of 4,096 for Cora, Citeseer, and Pubmed, and 16,384 for Reddit. For Reddit, BoostGCN reports a 21% reduction in external memory access for the tile size of 16,384 over the tile size 1,024.

One thing we can further improve is the implementation of double buffering, i.e., the ping-pong strategy to hide the latency in data movement from off-chip memory. Both ASAP2020 and BoostGCN use this strategy for acceleration. They also use dedicated cores for transformation and aggregation. Compared to ASAP2020 and BoostGCN, our improvement mainly comes from the reconfigurable core architecture, which helps overcome the pipeline stall, and the design with the symmetry property, which enables us to produce the same number of outputs by using approximately half the number of compute tiles. In the future, we plan to add double buffering since we still have enough memory resource.

Finally, we evaluate our result with HyGCN [3], a ASIC based implementation in TSMC 12nm CMOS. It has 4,096 32-bit fixed-point multipliers and 512 32-bit fixed-point ALUs running at 1 GHz, as compared to our FPGA design with 3,960 DSPs and at 220MHz. Despite having a vast difference in terms of computing resources and frequency, our FPGA design outperforms HyGCN for Citeseer and Reddit, by 3× and 3.9×, respectively. This is because HyGCN does not consider the inherent property of the graph, such as the symmetry property, which limits its performance for bigger datasets.

V. CONCLUSION

Sparse feature aggregation is the primary bottleneck in the acceleration of GCNs. In this paper, we leverage the symmetry property of the adjacency matrix in an undirected graph to accelerate feature aggregation, achieving end-to-end GCN acceleration. We propose a custom dataflow based on the symmetry property and design a hardware accelerator architecture to support the dataflow. We pair this architecture with a software-based pre-possessing method to maximize data locality and to reduce memory access. Our design achieves up to 1,101×, 17×, 110.5× improvement over CPU, GPU and other FPGA implementations. Further improvement can be achieved through the implementation of double buffering and lower-precision in PE design.

REFERENCES

[1] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[2] Tong Geng et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *IEEE/ACM MICRO*, 2020.

[3] Mingyu Yan et al. Hygcn: A GCN Accelerator with Hybrid Architecture. In *HPCA*. IEEE, 2020.

[4] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[5] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[6] Bingyi Zhang et al. Boostgcn: A framework for optimizing GCN inference on FPGA. In *IEEE FCCM*, 2021.

[7] Bingyi Zhang et al. Hardware acceleration of large scale GCN inference. In *IEEE ASAP*, 2020.

[8] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[9] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.