SkeletonGCN: A Simple Yet Effective Accelerator For GCN Training

Chen Wu, Zhuofu Tao, Kun Wang, and Lei He*,
Electrical and Computer Engineering, University of California, Los Angeles, USA

***Abstract*—Graph Convolutional Networks (GCNs) have shown great results but come with large computation costs and memory overhead. Recently, sampling-based approaches have been proposed to alter input sizes, which allows large GCN workloads to align to hardware constraints. Motivated by this flexibility, we propose an FPGA-based GCN accelerator, named SkeletonGCN, along with multiple software-hardware co-optimizations to improve training efficiency. We first quantize all feature and adjacency matrices of GCN from FP32 to SINT16. We then simplify the non-linear operations to better fit the FPGA computation, and identify reusable intermediate results to eliminate redundant computation. Moreover, we employ a linear time sparse matrix compression algorithm to further reduce memory bandwidth while allowing efficient decompression on hardware. Finally, we propose a unified hardware architecture to process sparse-dense matrix multiplication (SpMM) and dense matrix multiplication (MM), all on the same group of PEs to increase DSP utilization on FPGA. Evaluation is performed on a Xilinx Alveo U200 board. Compared with existing FPGA-based accelerator on the same network architecture, SkeletonGCN can achieve up to 11.3x speedup while maintaining the same training accuracy. In addition, SkeletonGCN can achieve up to 178x and 13.1x speedup over state-of-art CPU and GPU implementation on popular datasets, respectively.**

***Index Terms*—GCN Training Accelerator, Fixed-point Quantization, Unified Architecture, SpMM.**

I. Introduction

Over recent years, graph convolutional networks (GCNs) [1] have become popular solutions in real-world applications, such as drug development and web-scale recommenders [2], [3].

However, the promising performance comes at the cost of enormous memory and computation burden. GCN is often used to process large graphs, whose size ranges into gigabytes and have trouble fitting onto hardware such as GPUs and FPGAs. Fortunately, there have been sampling-based approaches [4], [5] to decompose large graphs into smaller sub-graphs, allowing them to fit on hardware while maintaining training accuracy. On the other hand, both memory consumption and computation efficiency can be further reduced through quantization, reducing representation down to INT8 with little accuracy loss [6]. However, there is yet to be an accelerator to fully leverage the advantages of the approaches above.

CNNs accelerators [7], [8] explore parallel computing to improve efficiency by depending heavily on data regularity and re-usage. However, it is challenging to apply such techniques to accelerate GCN training [1], [4], [5] due to the sparsity in graph adjacency matrices. This sparsity incurs a sparse-dense matrix-matrix multiplication (SpMM), which results in irregular memory access, as well as PE imbalance. Moreover, GCNs also incur a dense matrix-matrix multiplication (MM), which requires heavy arithmetic resources.

There exist GCN accelerators to overcome the above challenges during inference. AWB-GCN [9] processes SpMM and MM in pipelined modules, and uses a dynamic allocation scheme to keep each module actively utilized. EnGN [10] proposes a uniformed architecture to accelerate SpMM and MM. However, expanding to training is difficult as they either cache large amounts of data on-chip or depend on high bandwidth memory. During training, most intermediate results need to be stored for backward, thus requiring much larger memory capacities. GraphACT [11] proposes a CPU-FPGA heterogeneous platform to accelerate GCN training, leveraging a redundancy reduction algorithm. However, they implement the feature propagation and weight transformation as separate modules, which strays away from a uniform architecture may lead to low hardware efficiency.

To this end, we propose **SkeletonGCN**, a simple yet effective FPGA-based accelerator for GCN training with both algorithm and hardware optimizations. We first apply quantization to reduce storage requirement, we then simplify the non-linear operations to better fit FPGA. Thereafter, to further reduce storage and bandwidth consumption, we improve the PCOO format proposed in LW-GCN [12] by removing redundant information by proposing compact PCOO (CPCOO). A unified PE architecture is then developed to efficiently handle SpMM, MM and MM with transposed input (referred to as TMM below). The architecture is fully pipelined and equipped with ping-pong buffers to maximize DSP efficiency. We evaluate on a Xilinx Alveo U200 board. The experimental results show that our simplification steps incur negligible accuracy loss, while providing significant efficiency boost.

To summarize, our main contributions are as follows:

- **Simple yet effective training:** We simplify computation through compressed scheduling, quantization, and reused intermediate results. Experimental results show that **SkeletonGCN** offers comparable training accuracy despite these proposed simplification techniques.
- **Unified high efficiency PE architecture:** It supports SpMM, MM and TMM with high DSP efficiency. Comprehensive experiments show that we can achieve up to 95% DSP efficiency, which in turn contributes to overall low training latency.

*Corresponding author: Lei He (lhe@ee.ucla.edu).

- **Low training latency:** Compared with a prior FPGA-based accelerator [11] under the same experiment settings, **SkeletonGCN** achieves up to 11.3× speedup for total training convergence time. The speedup is up to 178× and 13.1× respectively compared with state-of-the-art CPU and GPU.

II. Background and Related Work

A. Workload Breakdown

Following the state-of-the-art architectures, GCN [1] and GraphSAINT [5] include multiple graph convolution layers or multi-layer perceptron (MLP) layers. The training process includes forward propagation, backward propagation, and weight update. To avoid ambiguity, the mathematical definitions used throughout this work are listed as follows.

Forward propagation in GCN is defined in Equ. 1.

$$X_l = \text{ReLU}(AX_{l-1}W_l), \tag{1}$$

where X_{l-1} and X_l denote the feature matrix of layer $l-1$ and l, respectively. W_l denotes the weight matrix of layer l while A is the adjacency matrix of the input graph. Since A is always sparse while X and W are typically dense, the basic operations for the forward phase are SpMM and MM. GraphSAINT introduces a concept of **order**. An order 0 layer is simply an MLP layer, while an order 1 layer is as follows:

$$X_l = \text{ReLU}(\begin{bmatrix} X_{l-1}W_{l,a} & AX_{l-1}W_{l,b} \end{bmatrix}), \tag{2}$$

where the [.] indicates column-wise concatenation, and $W_{l,a}, W_{l,b}$ are the self and neighbor weight matrices of layer l, respectively. The MLP layer performs a simple MM as $X_L = \sigma(X_{L-1}W_L)$. GraphSAINT also inserts an L2 normalization before the final MLP layer, as shown in Equ. 3.

$$X_l = \frac{X_{l-1}}{||X_{l-1}||_2}. \tag{3}$$

Gradients are computed during backward propagation before weights are updated via the Adam optimizer [13]. The gradients of layer $l-1$ for GCN are shown in Equ. 4 and 5.

$$\frac{\partial \mathcal{L}}{\partial X_{l-1}} = \mathbb{1}_{X_{l-1}>0}[W_{l-1}^T A^T \frac{\partial \mathcal{L}}{\partial X_l}], \tag{4}$$

$$\frac{\partial \mathcal{L}}{\partial W_{l-1}} = A^T \frac{\partial \mathcal{L}}{\partial X_l} X_{l-1}^T, \tag{5}$$

where the superscript T denotes matrix transpose. Backward phase also mostly consists of SpMM and MM. However, the input to an MM may be a transposed copy of a previous feature map or weight, which requires a different data access pattern, we refer to this case as TMM. The gradients of layer $l-1$ for GraphSAINT are computed as follows:

$$\begin{bmatrix} \frac{\partial \mathcal{L}}{\partial X_{l,a}} & \frac{\partial \mathcal{L}}{\partial X_{l,b}} \end{bmatrix} = \frac{\partial \mathcal{L}}{\partial X_l}, \tag{6}$$

$$\frac{\partial \mathcal{L}}{\partial X_{l-1}} = \mathbb{1}_{X_{l-1}>0}[W_{l-1,a}^T \frac{\partial \mathcal{L}}{\partial X_{l,a}} + W_{l-1,b}^T A^T \frac{\partial \mathcal{L}}{\partial X_{l,b}}] \tag{7}$$

$$\frac{\partial \mathcal{L}}{\partial W_{l-1,a}} = \frac{\partial \mathcal{L}}{\partial X_{l,a}} X_{l-1,a}^T \tag{8}$$

$$\frac{\partial \mathcal{L}}{\partial W_{l-1,b}} = A^T \frac{\partial \mathcal{L}}{\partial X_{l,b}} X_{l-1,b}^T \tag{9}$$

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial X_{l-1}} &= \frac{\frac{\partial \mathcal{L}}{\partial X_{l-1}}}{||X_{l-1}||_2} - X_{l-1} \frac{\sum(X_{l-1} \times \frac{\partial \mathcal{L}}{\partial X_{l-1}})}{||X_{l-1}||_2^3} \\ &= \frac{\frac{\partial \mathcal{L}}{\partial X_{l-1}}}{||X_{l-1}||_2} - \frac{X_{l-1}}{||X_{l-1}||_2} \sum (\frac{X_{l-1}}{||X_{l-1}||_2} \times \frac{\frac{\partial \mathcal{L}}{\partial X_{l-1}}}{||X_{l-1}||_2}) \end{aligned} \tag{10}$$

In Equ. 4 - 10, $\mathcal{L}$ denotes the training loss. The [.] on the left side of Equ. 6 indicates a column-wise partitioning into two blocks with equal number of columns. After computing the gradients, the weights are updated via the Adam Optimizer [13].

For all input (sub)graphs in GraphSAINT related computation, we directly use the random-walk sampler as proposed in the original text [5]. The number of roots per sub-graph is tuned per dataset to generate sub-graphs with approximately 2048 nodes in order to fit our hardware design.

B. Low Precision Training

Various low precision training algorithms have been proposed to train CNNs [14]–[16]. However, these studies cannot directly be applied to GCNs training due to the input sparsity. There have been GNN quantization methods in [6], [17], which use a fine-tuning scheme to compensate for the accuracy loss. Despite their success in reducing memory requirements for GCN inference, the studies on quantization of large graphs and hardware-aware quantization is not sufficient.

C. FPGA-based Accelerators

FPGA-based accelerators have been explored extensively on CNN inference and training [8], [18]–[21]. The work in [8], [18], [19] uses the FPGA overlay technique to accelerate CNN inference with different data representations. CNN training accelerators are developed in these studies [20], [21] to accelerate mini-batch training on CNNs, and use HBM to meet external bandwidth constraints.

Although GCNs share a similar layer-based network architecture as CNNs, accelerating GCNs is different. Existing accelerators [9], [10] achieve speed boosts on GCN inference, however, their designs rely heavily on caching input and intermediate data on chip. This would be impractical for training, as much larger amounts of intermediate results must be stored for back-propagation. GraphACT [11] accelerates GCN training through redundancy reduction in software and parallel computing in hardware. HP-GNN [22] proposes a framework to generate GNN accelerators on a CPU-FPGA platform automatically. They reduce the memory traffic and random memory access to accelerate GNN training. However, both [11] and [22] implement SpMM and MM in separate modules, which reduces DSP runtime utilization.

III. Optimized Training

A. Training Simplifications

In this work, we perform the majority of GCN training with 16-bit signed integers (SINT16). We chose signed integer

TABLE I
DIMENSIONS, DENSITIES AND WORKLOADS ACROSS DATASETS.

Datasets	Nodes	Edges	Features	Classes	Edge Density
PPI	14755	225270	50	124	0.1035%
Reddit	232965	11606919	602	41	0.0214%
Yelp	716847	6977410	300	100	0.0014%

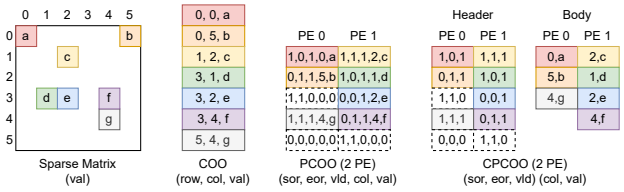

Fig. 1. CPCOO: Compact PCOO

representation over floating point as arithmetic operations with integers consume less hardware resources and power. The precision of 16 bits is selected as Xilinx provides native IP support in DSP configuration. While SINT8 is also supported, our experiments show that quantization to SINT8 would result in a significant accuracy degradation.

We apply quantization to the initial input feature maps and adjacency matrices, and initialize trainable parameters directly as SINT16. Experiments show that for most of computation, quantization yields negligible loss to final training accuracy, and would not affect convergence time in terms of epochs. However, for some non-linear operations including L2 normalization, softmax, and square root, applying quantization would devastate results, and they must be kept in FP32.

Due to the representation precision of SINT16, multiplication with a number within $[1 \pm 2^{-16}]$ results in no change at all. This allows two simplifications to the Adam weight update procedure. First, we eliminate the need to compute $\hat{m}_t$ and $\hat{v}_t$, as the quantized results are identical to the original m_t and v_t starting from the second to third epoch. Second, the learning rate η is rounded to the nearest power of 2 to replace the multiplication by a simple bit shift.

Furthermore, we detect intermediate results used in multiple places and cache them to prevent redundant computation. Specifically, the step $A^T \frac{\partial \mathcal{L}}{\partial X_l}$ is used twice in back propagation, and $||X_{L-1}||_2$ is used multiple times in L2 normalization and its gradient.

B. Hardware-aware Compression

As shown in Table I, adjacency matrices tend to be extremely sparse and would consume an impractical amount of storage, while less than 1% are non-zero elements. Therefore, it is necessary to only store and compute the non-zero elements.

We improve the PCOO format proposed in [12] to further compress the sparse matrices, since PCOO injects empty elements for empty rows. We propose the compact PCOO (CPCOO) by dividing PCOO into **header** and **body** parts, as shown in Fig. 1. The header field includes the SOR, EOR and VLD signals while the body field includes the column and value. The representation of the rows with non-zero elements is the same as PCOO. For empty rows, the header would contain VLD=0, and the body is skipped. During decompression, the hardware fetches the body according to the VLD signal provided in the header field (see details in Section IV-B). Since the data width for the body is much larger than the header, we can significantly reduce storage consumption for empty rows of large datasets. Table III-B shows the storage consumption of PCOO versus CPCOO per average sub-graph across three datasets. CPCOO reduces memory by $2.87\times$ to $4.81\times$ over PCOO.

TABLE II
COMPRESSED MATRIX SIZES ACROSS DATASETS AND ALGORITHMS (RANDOM-WALK SAMPLED SUB-GRAPHS [5]). ALL EVALUATED AS AVERAGE SIZE ACROSS DIFFERENT SUB-GRAPHS.

Dataset	Nodes	Edges	COO	PCOO	CPCOO
PPI	1992	41939	2.01Mb	4.89Mb	1.76Mb
Reddit	1977	10780	517Kb	1.65Mb	486Kb
Yelp	1959	7561	363Kb	1.98Mb	412Kb

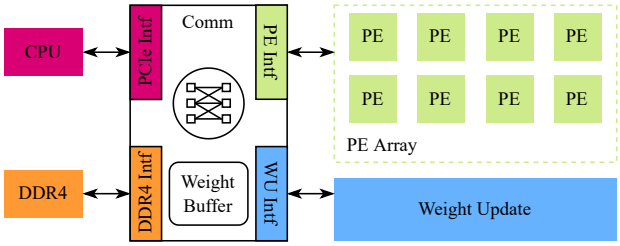

Fig. 2. Overall architecture of **SkeletonGCN**.

IV. HARDWARE ARCHITECTURE

In this section, we discuss in detail the hardware architecture of the proposed **SkeletonGCN**, which efficiently supports the training process of quantized GCN.

A. Overall Architecture

The FPGA performs the SpMM, MM, and TMM in forward and backward propagation, as well as element-wise operations in weight update. On the other hand, we assign softmax and categorical cross-entry loss to the CPU because they are hardware expensive to compute accurately. Graph sampling and data preprocessing are also assigned to the CPU. The detailed scheduling between CPU and FPGA will be discussed in Section IV-E.

Regarding workload assignment, the overall architecture of **SkeletonGCN** is shown in Fig. 2. The *PE Array Module* performs all the operations in the forward and backward phases, while the *Weight Update Module* is followed to update weights after back propagation. The *Communication Module* is responsible for the data transfer between CPU, FPGA, and off-chip memory (DDR4), and also among different PEs.

B. Unified PE Architecture

Since the whole graph is sampled and only one sub-graph is trained each time, we set the sub-graph size appropriate so

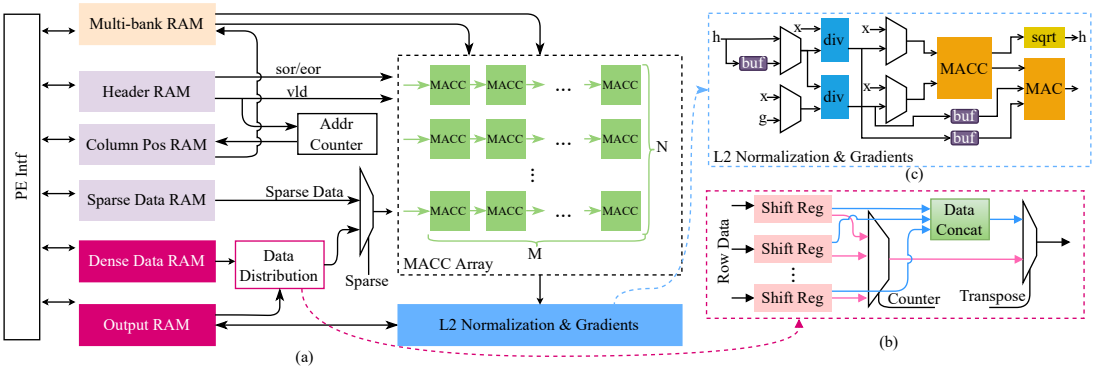

Fig. 3. (a) The architecture of the unified PE. (b) The architecture of the L2 normalization and gradients module. (c) The architecture of the data distribution module.

that all the data of a sub-graph can be handled by the on-chip memories of the FPGA. In this way, the FPGA only need to access the off-chip memory three times during the training of one sub-graph: 1) to get the initial data for the forward pass, 2) to send back the results of the last layer to compute loss and gradients on CPU, and 3) to get the gradients of the last layer for back-propagation.

To increase computation efficiency, we propose a unified PE architecture to perform the forward and backward phases, as shown in Fig. 3 (a). As discussed in Section III, the main operations are SpMM, MM and TMM, where the basic operation is multiply-accumulate (MACC). In this way, we first design an $M \times N$ *MACC Array* to support each SpMM, MM and TMM in parallel. Generally, all the MACC units in each row of the *MACC Array* share a same input. The challenge then becomes how to feed data into the MACCs to make it perform efficiently under different workloads.

1) SpMM: As discussed in Section III, we use the CPCOO format to compress sparse matrices and store the compressed data into 3 separate RAMs, as shown in Fig 3 (a). During computation, the Header is flushed out from the *Header RAM* and we use "VLD" signal to enable the *Address Counter*, which is used to generate the address of the *Body RAM*. The output of the *Body RAM* is the column position of the non-zero element, thus indicating the address of the corresponding dense data in the *Multi-bank RAM*. In this way, the decompression logic of data under the CPCOO format can be as simple as several wires and a counter, as shown in Fig 3 (a). The non-zero sparse data and corresponding dense data are then fed into the *MACC Array*, and the "SOR" and "EOR" signals control when to start computation for a new row and when to save the results. With the fully pipelined architecture, the MACC units keep active during most cycles of computing SpMM, thus leading to a high DSP efficiency (see details in Section V-D).

2) MM and TMM: Although MM and TMM share the same computation operations, they require different memory access for matrices and transposed matrices. To save memory access burden and improve DSP efficiency, we design a uniform memory load and store logic for both MM and TMM. In contrast, a *Data Distribution Module* is added to control the data needed by the *MACC Array* for computing MM and TMM, as shown in Fig. 3 (b). The row data of the left matrix is first fetched from the on-chip memory and then fed into the shift registers. Each shift register stores one row and can be configured to output one element or the entire row according to computation mode. When computing MM, each shift register outputs one element and all the elements are concatenated and fed into the N rows of the *MACC Array* (blue arrow in Fig. 3 (b)). On the other hand, when computing TMM, the shift registers output the entire row one by one and feed into the N rows of the *MACC Array* (pink arrow in Fig. 3 (b)). By setting the data width of the on-chip memory $N \times N \times 16$ (we use SINT16), we can keep outputting active data every cycle for both MM and TMM to make MACCs active, thus maintaining high DSP efficiency. Since we set $N = 16$ in our accelerator, the data width of $N \times N \times 16$ is easy to achieve by using BRAMs or URAMs in Xilinx FPGA.

3) L2 normalization and its gradients: We develop an extra module to process L2 normalization and its gradients, which takes the results of the *MACC Array* as inputs to pipeline the computation. We use the CORDIC IP and division IP in Xilinx FPGA to compute square root and division, respectively. As the L2 normalization and its gradients are computed in serial, we reuse most of the computation units to save resource utilization, as shown in Fig. 3 (c). All the multiplexers in Fig. 3 (c) are selected by the signal indicating the computation of L2 normalization or gradients. For the data which will be used in both L2 normalization and its gradients, we will buffer them to eliminate redundant computation.

C. Weight Update

Since the weights are updated after computing the gradients by TMM, we design a separate module that takes the outputs of the PE array to fully pipeline the computation, as shown in Fig. 2. The square root and division operations in Adam are also computed by using CORDIC and Division IPs. Moreover, we define the hyper parameters in Adam to the nearest power of 2 to simplify the multiplications to shift.

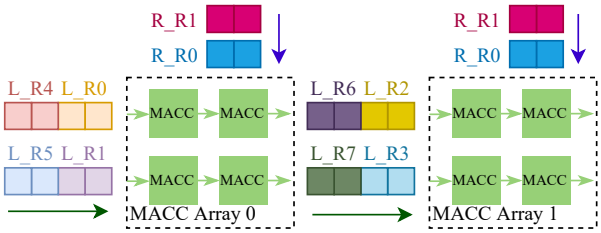

Fig. 4. An example of allocating SpMM, MM or TMM onto 2 *MACC Arrays*. "L_R#" indicates the row index of the left matrix while "R_R#" indicates the row index of the right matrix.

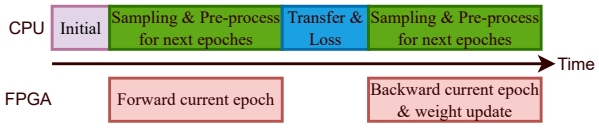

Fig. 5. Scheduling between CPU and FPGA.

TABLE III
RESOURCE UTILIZATION ON ALVEO U200 BOARD.

Resource	LUT	LUTRAM	BRAM	URAM	DSP
Used	1021386	183191	1338	598	4460
Available	1182240	591840	2160	960	6840
Utilization(%)	86.39	30.95	61.94	62.29	65.20

D. Data Communication

The *Communication Module* handles data transfer between external memory and FPGA and among different PEs, as shown in Fig. 2. The interface between CPU and FPGA is PCIe Gen3 X16, and only the initial data, final results of forward phase and first gradients for backward phase are transferred via PCIe. DDR4 is also used as the external memory to save initial data for different training epochs. The on-chip buffers are designed in ping-pong manner, so that the communication time between DDR4 and FPGA can be hidden under the computation time. Since the computation of one layer is allocated in parallel on different PEs, we also transfer data among different PEs. Moreover, we also collect all the data of one layer and input into the *Weight Update Module* for updating weights after back-propagation. Considering the properties of FPGA (*i.e.,* constraints of the number of long connections between different super logic regions), we use FIFOs in the *PE Interface Modules* to control the bandwidth between different PEs.

E. Allocation and Scheduling

1) Allocation: For SpMM, MM and TMM, we use a round robin method to assign different rows of the left matrix onto different rows of different PEs, as shown with a simple example in Fig. 4. In this way, we can hide the row information of the non-zero elements under the row index of the *MACC Array*, thus simplifying CPCOO. Moreover, the elements in the left matrix are fed into the MACCs one by one (green arrow) and the elements of the right matrix are fed into the PEs row by row (blue arrow). Since we are not able to perform the whole matrix, we do a row-wise partition in the left matrix and a column-wise partition in the right matrix to fit one tile into the *MACC Array*. In this way, each PE can only achieve a portion rows of the result matrix. When the result matrix is used as the left matrix in next steps, we will will propagate the results inside each PE. Otherwise, we will communicate among different PEs to collect the whole result matrix.

2) Scheduling: During the training of GCN, the computation expensive operations are assigned to the FPGA while others are assigned to CPU. To improve the overall training performance, we parallel most of the operations between CPU and FPGA, as shown in Fig. 5. After the first data initialization, the CPU keeps on sampling and preprocessing data for the next epochs while the FPGA is forwarding the current epoch. The CPU is interrupted to transfer forward results and compute the softmax, loss and the corresponding gradients once the forward phase is finished. The gradients are then transferred back to the FPGA for backward and the CPU is back to preparing data for next epochs. In addition, the multi-core CPU can be set to work in parallel because the sub-graphs are independent to each other.

V. EXPERIMENTAL RESULTS

In this section, **SkeletonGCN** is evaluated with comprehensive experiments. We first evaluate the proposed SINT16 training approach on different datasets and networks to show its effectiveness. The FPGA accelerator is then evaluated on GraphSAINT with different configurations. Finally, we compare our work with GraphACT [11] under the same FPGA configurations.

A. Experimental Setup

SkeletonGCN is implemented with Verilog HDL and deployed on a Xilinx Alveo U200 board. We implement 8 PEs, each of which is equipped with a 32×16 *MACC Array* for SpMM, MM and TMM. After synthesis and implementation with Vivado 2020.1, the overall resource utilization is shown in Table V-A.

The datasets used in our experiments are shown in Table I. We take the the same sampling algorithms as GraphSAINT and GraphACT for fair comparison. All the results on CPU and GPU are produced from PyTorch Geometric [23] and the open-sourced code provided by GraphSAINT [5].

B. Training Accuracy and Latency

Fig. 6 shows GraphSAINT [5] training accuracy across three popular datasets. The original GraphSAINT configuration uses larger sub-graphs (*i.e.,* 8000 nodes) as proposed by the original authors. The simplified version uses 2000 node sub-graphs to better fit our hardware, and removes a portion of functionality including dropout and batch normalization. The quantized version is ran on our hardware, mostly in SINT16 as described in previous sections. Results show insignificant drops in F1 score of approximately 0.5-0.7% for Reddit and Yelp datasets.

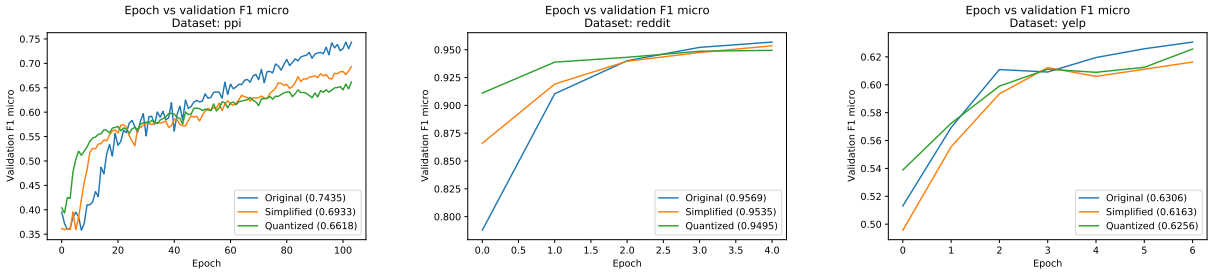

Fig. 6. Training Accuracy Comparison

TABLE IV
COMPARISON WITH GRAPHACT, CPU, GPU ON GCN AND GRAPHSAINT. "-" INDICATES NO REPORTED RESULTS, "*" INDICATES RESULTS DIRECTLY TAKEN FROM GRAPHACT [11]

		GraphACT	CPU	GPU	Ours
Data type		FP32	FP32	FP32	SINT16
Frequency(GHz)		0.2	2.2	1.2	0.25
DSP/CPU/Cuda		5632	40	3584	4460
Total convergence time on GraphSAINT (s)	PPI	-	352.5	8.3	7.1
	Reddit	-	72.5	2.9	0.96
	Yelp	-	965.1	27.7	27.1
Total convergence time on GCN (s)	PPI	9.6*	151.4*	10.6*	0.85
	Reddit	7.6*	95.5*	11.4*	0.87
	Yelp	23.4*	359.4*	30.4*	3.76

The drop for PPI was significant at 8% because the PPI dataset is less robust to smaller sub-graph sizes [5].

The training latency under the same configurations is shown in Table IV. On average, we achieve 53.5× and 1.7× speedup compared with CPU and GPU, respectively.

C. Comparison with State-of-the-art

We also compare our work with GraphACT [11] on the same experimental settings for fair comparison. The GCN evaluated has two graph convolution layers and one MLP layer, and the hidden size is set to 256 for all graph convolution layers. As shown in Table IV, we achieve speedup of up to 11.3× compared with GraphACT across all datasets.

The advantages come from both our quantization-aware training algorithm and our unified PE architecture. First, the SINT16 representation greatly reduces the usage of DSPs (in Xilinx FPGA, each FP32 multiplier consumers 3 DSPs while each SINT16 multiplier consumers 1 DSP). Second, we use the unified PE architecture, which dramatically increase the DSP efficiency.

D. Discussion

This section discuss the DSP efficiency, defined as follows:

$$DSP\_EFF = \frac{Lat_{theo}}{Lat_{test}}, \tag{11}$$

where the Lat_{theo} and Lat_{test} indicate the theoretical latency and tested latency, respectively. We skip all the zeros in SpMM and the theoretical latency of SpMM and MM are then calculated by using Equ. 12 and 13.

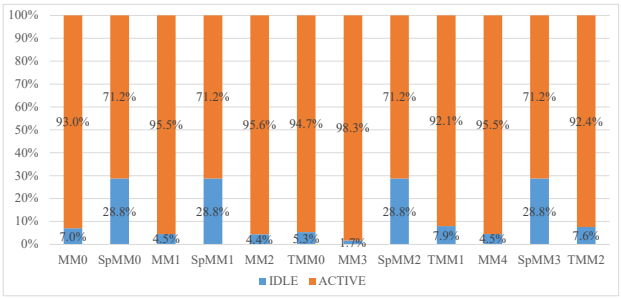

Fig. 7. DSP efficiency of SpMM, MM, and TMM in training GCN on Reddit.

$$Lat_{theo}^{SpMM} = \frac{\#\ of\ non-zero\ MAC\ ops}{\#\ of\ MAC\ units}. \tag{12}$$

$$Lat_{theo}^{MM} = \frac{\#\ of\ MAC\ ops}{\#\ of\ MAC\ units}. \tag{13}$$

We analyze the average DSP efficiency of training GCN on Reddit, as shown in Fig. 7. We can see that DSP efficiency for computing MM and TMM can be up to 98.3%. The DSP efficiency of SpMM is 71.2% because we inject empty elements to avoid bank conflicts as that of the PCOO format. However, it has little influence on the total training latency because SpMM only accounts for around 1% of the total computation workloads.

VI. CONCLUSION AND FUTURE WORK

In this work, we propose **SkeletonGCN** to improve GCN training efficiency on FPGA. The data representation is first quantized to SINT16 to reduce computation and storage requirements. In addition, we simplify the non-linear operations and eliminate redundant computations to better fit the computation on FPGA. Moreover, we employ CPCOO format to further compress sparse matrices while allowing efficient decompression on hardware. A unified hardware architecture is then proposed to compute SpMM, MM and TMM to improve DSP efficiency. Evaluation shows that our simplified training approach can train the network with negligible accuracy loss. Moreover, **SkeletonGCN** can achieve up to 11.3× speedup over existing FPGA-based accelerator and up to 178× and 13.1× speedup over state-of-the-art CPU and GPU, respectively.

REFERENCES

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

[2] M. Sun, S. Zhao, C. Gilvary, O. Elemento, J. Zhou, and F. Wang, "Graph convolutional networks for computational drug development and discovery," *Briefings in bioinformatics*, vol. 21, no. 3, pp. 919–935, 2020.

[3] 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.

[4] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 257–266.

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

[6] S. A. Tailor, J. Fernandez-Marques, and N. D. Lane, "Degree-quant: Quantization-aware training for graph neural networks," *arXiv preprint arXiv:2008.05000*, 2020.

[7] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "Opu: An fpga-based overlay processor for convolutional neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 35–47, 2020.

[8] Y. Yu, T. Zhao, K. Wang, and L. He, "Light-opu: An fpga-based overlay processor for lightweight convolutional neural networks," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 122–132. [Online]. Available: https://doi.org/10.1145/3373087.3375311

[9] 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.

[10] S. Liang, Y. Wang, C. Liu, L. He, L. Huawei, D. Xu, and X. Li, "Engn: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Transactions on Computers*, vol. 70, no. 9, pp. 1511–1525, 2020.

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

[12] Z. Tao, C. Wu, Y. Liang, and L. He, "Lw-gcn: A lightweight fpga-based graph convolutional network accelerator," *ArXiv*, vol. abs/2111.03184, 2021.

[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[14] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, "Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[15] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742.

[16] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 8612–8620.

[17] B. Feng, Y. Wang, X. Li, S. Yang, X. Peng, and Y. Ding, "Sgquant: Squeezing the last bit on graph neural networks with specialized quantization," in *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2020, pp. 1044–1052.

[18] C. Wu, M. Wang, X. Chu, K. Wang, and L. He, "Low-precision floating-point arithmetic for high-performance fpga-based cnn acceleration," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 1, nov 2021. [Online]. Available: https://doi.org/10.1145/3474597

[19] C. Wu, J. Zhuang, K. Wang, and L. He, "Mp-opu: A mixed precision fpga-based overlay processor for convolutional neural networks," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 33–37.

[20] S. Kolala Venkataramanaiah, Y. Ma, S. Yin, E. Nurvithadhi, A. Dasu, Y. Cao, and J.-S. Seo, "Automatic compiler based fpga accelerator for cnn training," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 166–172.

[21] S. K. Venkataramanaiah, H.-S. Suh, S. Yin, E. Nurvitadhi, A. Dasu, Y. Cao, and J.-s. Seo, "Fpga-based low-batch training accelerator for modern cnns featuring high bandwidth memory," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3400302.3415643

[22] Y. Lin, B. Zhang, and V. K. Prasanna, "HP-GNN: generating high throughput GNN training implementation on CPU-FPGA heterogeneous platform," *CoRR*, vol. abs/2112.11684, 2021. [Online]. Available: https://arxiv.org/abs/2112.11684

[23] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.