Title
FPGA-Based Graph Convolutional Neural Network Acceleration

Author
TAO, ZHUOFU

Publication Date
2022

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

FPGA-Based Graph Convolutional Neural Network Acceleration

A thesis submitted in partial satisfaction

of the requirements for the degree Master of Science

in Electrical and Computer Engineering

by

Zhuofu Tao

2022

ABSTRACT OF THE THESIS

FPGA-Based Graph Convolutional Neural Network Acceleration

by

Zhuofu Tao

Master of Science in Electrical and Computer Engineering

University of California, Los Angeles, 2022

Professor Lei He, Chair

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, this thesis proposes an FPGA-based GCN accelerator, along with a novel sparse matrix format and multiple software-hardware co-optimizations to improve training efficiency. First, all feature and adjacency matrices of GCN are quantized from 32-bit floating point to 16-bit signed integers. Next, the non-linear operations are simplified to better fit the FPGA computation, and reusable intermediate results are identified and stored to eliminate redundant computation. Moreover, a linear-time sparse matrix compression algorithm is employed to further reduce memory bandwidth, while allowing efficient decompression on hardware. Finally, a unified hardware architecture is proposed to process sparse-dense matrix multiplication (SpMM), dense matrix multiplication (MM) and transposed matrix multiplication (TMM), all on the same group of PEs to maximize DSP utilization on FPGA.

Evaluation is performed on a Xilinx Alveo U200 board. Compared with existing FPGA-

based accelerator on the same network architecture, the new accelerator achieves up to 11.3× speedup while maintaining the same training accuracy. It also achieves up to 178× and 13.1× speedup over state-of-art CPU and GPU implementation on popular datasets, respectively.

The thesis of Zhuofu Tao is approved.

Corey Wells Arnold

Lin Yang

Lei He, Committee Chair

University of California, Los Angeles

2022

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Deep learning [5] have been one of the most popular research topics over recent years, and have achieved amazing results across numerous fields. The powerful adaptability of data-driven approach allows researchers to develop effective solutions for arbitrary problems without having much expertise on them. In the earlier days of deep learning, vanilla deep neural networks fail to achieve high performance on fields such as natural language processing. However, the field adapted by proposing domain-specific variations of neural networks for specific data formats. The introduction of convolutional neural networks (CNNs) [6] to aggregate local data and recurrent neural networks (RNNs) [9] to track state over time were both successful, and inspired many researcher to propose new variations to the algorithm.

Graph-based neural networks are a particularly interesting family of models, in the sense that they eliminated a fundamental dependency on Euclidean data, which still exist in most other deep learning models. The term "Euclidean" data refers to the ability to represent a data tensor in a multi-dimensional Euclidean space. For most problems, individual cases are examined separately, each case is represented by n features, which easily translate to an n-dimensional feature vector to satisfy the Euclidean constraint. However, in some problem settings, individual cases are highly correlated with each other and form a graph structure. As the number of neighbors of each node (i.e. its degree) is variant, graphs are inherently non-Euclidean, which poses a problem for traditional neural networks.

Motivated by this problem, researchers first proposed the idea of graph neural networks [10], which started as an abstract class of models that incorporate graph information in its computation. At first, the graph was only used as a regularization term to penalize differences between prediction output on neighboring nodes [17]. While this was already an improvement, the first direct use was not until much later, when T. Kipf et al. took inspiration from convolutional neural networks and applied convolution on the graph fourier

domain, and were able to propose the graph convolutional neural network (GCN) [4]. From this point on, graph information was commonly represented as adjacency matrices, and would directly participate in forward propagation as operands in matrix multiplications.

GCN and its variations [2, 12, 13] achieved marginally better prediction accuracy than previous methods, however, they pose a challenge to the underlying computer architecture that performs the computation. Previously, the two most common operations in neural networks were dense matrix multiplication (MM) and convolution, both of which provide many structural properties. The regular loop structures in MM and convolution allow easy workload scheduling and parallelization, which then easily exploit the powerful parallel computation capability in modern multi-core hardware. Although GCNs also process adjacency matrices in matrix multiplications, the problem is that adjacency matrices are inherently sparse. Many popular graph datasets come with adjacency matrices with over 99% sparsity, and real-world datasets can be even sparser. Naively performing this matrix multiplication would result in many additions and multiplications with zero, and would be extremely inefficient.

In order to acknowledge the sparsity in the adjacency matrix and perform a sparse-dense matrix multiplication (SpMM), the method would have to first preprocess the adjacency matrix into a compressed format, before tailoring data access and computation accordingly. There are several popular compressed sparse matrix formats such as compressed sparse rows (CSR), compressed sparse columns (CSC), and coordinate list (COO). Each of these formats can effectively skip zero elements in sparse matrices, and their storage consumption is linear to the number of non-zero elements. However, these formats result in irregular locations for non-zero elements, which complicates data load operations. For example, in order to read the first non-zero element of an arbitrary row in a CSR representation, the processor must first read the start location of said row from the row indices vector, before translating the index into an address to read the actual element. This complication often results in

inefficient execution, and hinders the performance gains achieved by the compressed method in the first place.

Furthermore, again due to the extreme sparsity of adjacency matrices, the other operand matrix in the SpMM only gets to participate in a limited number of computation. In other words, the data reuse rate is often as low as only 1-3 floating point operations per byte (FLOPs/B), well below the data-bandwidth-compute ratio offered at DRAM level. This results in a memory-bound problem. Although smaller caches usually provide higher bandwidth and may support the low data reusage, their capacity is also very limited. Many popular graph datasets such as Reddit and Yelp come with data in the gigabyte range, and have trouble fitting on caches. Overall, it is difficult to construct a hardware architecture to efficiently process GCNs.

There have been GCN-specific accelerators that seek to tackle this problem, and each have achieved great improvements in their dedicated front. One of the earliest attempt was HyGCN [14], which considered SpMM and MM two different operations, and deployed each of them onto dedicated hardware modules to process. This approach opens the opportunity to split the compute units across modules, which reduces the data demand in each individual module. Furthermore, data pipeline across modules are constructed through on-chip SRAM, which generally provides a much higher bandwidth than off-chip DRAM, consequently alleviates the memory bound. While the advantage was solid, this approach had its own limitations as well. Despite the possibility to pipeline the two operations and keep all modules active, the design is not adaptable. The workload ratio between SpMM and MM depend directly on the dimensions and sparsity of input, which would vary across different datasets. FPGA-based hardware design takes a long time (i.e. several hours) to synthesize, it would be unfeasible to reconfigure a design and adjust module sizes to fit each new dataset. This problem would persist in other works later on [1], which process SpMM and MM on the same type of module, but still uses multiple pipelined modules across layers.

Additionally, new graph datasets are coming into attention with such large sizes that would not fit on-chip at all, and would cause further trouble for these methods.

Due to the large data sizes, most proposed accelerators would only target the inference phase of GCN computation [1, 7, 8, 14], as the training phase would require storing intermediate feature maps, posing more challenge to on-chip memory capacity. Interestingly, the large graph sizes inspired sampling-based solutions which break down graphs into smaller subgraphs. It have been shown in GraphSAINT [16] that with the proper normalization, training and inference with the subgraphs would lead to the same outcome as the full graph. This breakthrough resolved the capacity burden, and allowed the same authors to propose the first GCN training accelerator [15] at the time. Nevertheless, there remains room for improvement in terms of performance.

The GCN acceleration problem can be tackled from a different front. Since the most challenging step is the memory-bound SpMM operation, and must involve sparse matrix compression, it is sensible to optimize the compression format. This thesis will present a customization on the COO format, designed to simplify decompression during computation and resolve the need to access memory twice per non-zero element.

2 Background

This chapter will first describe the vanilla GCN architecture, as well as the sampling-based GraphSAINT architecture. This would naturally lead into workload scheduling on a FPGA-based hardware architecture, which would then motivate the novel compression format to be proposed in the next chapter.

2.1 Workload Breakdown

Graph convolutional network (GCN) [4] is first proposed as an approximation of spectral graph convolution. The network consists of multiple graph convolution layers, the forward propagation rule of which is shown in Equation 1:

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

where X_l and X_{l-1} denote the feature map on layer l and $l-1$ respectively, A denotes the graph adjacency matrix, and W_l denotes the trainable parameters on the lth layer, also known as the weight matrix. The layer number l is 1-indexed, and X_0 would denote the input feature map.

Since most graphs are not fully connected, the adjacency matrix A is sparse. In fact, Table 1 that the adjacency matrix in many popular graph datasets is at least 99% sparse. On the contrary, feature matrix X_{l-1} and weight matrix W_l are usually dense. The dimension of A, X_{l-1}, and W_l are $N \times N$, $N \times F_{l-1}$ and $F_{l-1} \times F_l$, where N denotes the number of nodes in the graph, and F_l denotes the feature dimension in layer l, respectively. Generally, in both full graphs and sampled subgraphs, the number of nodes N is larger than the number of input features F_0, and the number of features per layer is non-increasing with $F_{l-1} \geq F_l$.

Dataset	Nodes (N)	Edges (M) (Density)	Input Features (F_0)	Classes (F_L)
PPI	14755	225270 (0.1035%)	50	124
Reddit	232965	11606919 (0.0214%)	602	41
Yelp	716847	6977410 (0.0014%)	300	100

Table 1: Common graph datasets

These observations would then dictate the optimal operation order. Matrix multiplication is associative, and therefore $(AX_{l-1})W_l = A(X_{l-1}W_l)$, however, the number of floating point operations is not the same. In the two cases, the number of multiplications and additions required are $MF_{l-1}+NF_{l-1}F_l$ and $NF_{l-1}F_l+MF_l$, respectively. Here M denotes the number of edges in A. Since $F_{l-1} \geq F_l$ most of the time, it is generally favourable to first compute $X_{l-1}W_l$. In fact, in some other graph datasets, the input feature map X_0 can be sparse as well, computing AX_0 would result in a sparse-sparse matrix multiplication, which is much more difficult to perform efficiently.

The GraphSAINT [16] architecture, on the other hand, introduces several differences. Firstly, it introduces the concept of "order", and includes multiple graph convolution operations per layer. A layer with order o is defined in Equation 2:

$$X_l = \Big[X_{l-1}W_{l,0} \quad AX_{l-1}W_{l,1} \quad \dots \quad A^oX_{l-1}W_{l,o}\Big] \tag{2}$$

where [.] indicate the column-wise concatenation of intermediate results. Each layer would include multiple weight matrices from $W_{l,0}$ through $W_{l,o}$. In practice, the layer order used in GraphSAINT was either 0 or 1, and therefore a layer is either a dense layer in a multiple-layered perceptron (MLP) at order 0, or the column-wise concatenation of it and a vanilla graph convolutional layer at order 1.

GraphSAINT also inserts an L2 normalization operation at the second last layer, as

shown in Equation 3:

$$X'_{L-1} = \frac{X_{L-1}}{||X_{L-1}||_2} \tag{3}$$

where $||X_{L-1}||_2$ denotes the column-wise L2 norm of X_{L-1}.

For the training process of GCN and GraphSAINT, gradients are computed during backward propagation before weights are updated via the Adam optimizer [3]. The gradients of layer $l-1$ for GCN are shown in Equation 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. The basic operations during the backward phase are also 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, this case is termed "transposed matrix multiplication" (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 Equation 4 - 10, $\mathcal{L}$ denotes the training loss, specifically categorical cross entropy loss . The [.] on the left side of Equation 6 indicates a column-wise partitioning into two blocks with equal number of columns, opposite of the concatenation step in Equation 2. After computing the gradients, the Adam Optimizer is used to update weight matrices, as shown in Equation 11:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) \frac{\partial \mathcal{L}}{\partial W_{t-1}} \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) (\frac{\partial \mathcal{L}}{\partial W_{t-1}})^2 \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\ W_t &= W_{t-1} - \eta \frac{\hat{m}_t}{\hat{v}_t + \epsilon} \end{aligned} \tag{11}$$

where t denote the current epoch, m_t and v_t are persisted momentum variables throughout training, $\beta_1 = 0.9$ and $\beta_2 = 0.999$ are weight constants to balance between current epoch gradient vs previous epoch momentum, $\eta = 0.01$ is the learning rate, and $\epsilon = 10^-12$ is added to prevent division by zero. During actual computation, the $\hat{m}_t$ and $\hat{v}_t$ steps can be skipped without much effect on training accuracy or convergence time, and $\eta = 1/128$ is used instead to simplify the division to a bitshift.

2.2 Workload Scheduling On FPGA

MM is inherently structured with 3 rectangular loops, and is relatively simple to deploy onto any hardware architecture. Typically, in an FPGA-based design, numerical operations such as multiplications and additions are performed on digital signal processor (DSP) -based compute units. In order to maximize efficiency, these operations would typically be pipelined into several stages. Each multiplier or adder is capable of producing results at a throughput of 1 output per cycle, but at a latency of 5-20 cycles. Since the innermost loop of matrix multiplication is typically a dot product, individual multiplications over time are completely independent, but additions depend directly on the result of the previous additions. The adder would not produce the result of the current operation until several cycles later, and would block the next addition from dispatching immediately. This results in an initiation interval (II) of larger than 1, and is inefficient.

In order to achieve II=1 and eliminate idle time for compute units, accumulators are used instead of adders. This way, multiple inputs can be sequentially injected into the same accumulator over time, and the result can be read after the full dot product is complete. While this design eliminates idle time, it imposes a restriction that every dot product must be performed on the same accumulator, whereas previously any adder would be capable of performing the computation. In a matrix multiplication, dot products are performed with rows in the left input matrix and columns of the right input matrix. The following text will use the term "assign" for mapping a row from the left matrix onto a compute unit.

In an SpMM, the computation is slightly different. The zeros from the left input matrix must be skipped, and therefore the outer two loops from MM, as shown in Algorithm 1, are merged into a single loop, as shown in Algorithm 2. Consequently, the only remaining constant in the computation is number of columns (p) in the right input matrix. Since there is no data dependency between different columns, they can be spatially mapped to different

Algorithm 1: MM

inputs: $X \in \mathbb{R}^{m \times n}$, $W \in \mathbb{R}^{n \times p}$, $Y \leftarrow 0^{m \times p}$;
for *i in [0, m-1]* **do**
  for *j in [0, p-1]* **do**
    for *k in [0, n-1]* **do**
      $Y_{i,j} \leftarrow Y_{i,j} + X_{i,k} \times W_{k,j}$;
    end
  end
end
return Y

Algorithm 2: SpMM

inputs: $X \in \mathbb{R}^{m \times n}$, $W \in \mathbb{R}^{n \times p}$, $Y \leftarrow 0^{m \times p}$;
for $X_{i,k} \neq 0$ *in* X **do**
  for *j in [0, p-1]* **do**
    $Y_{i,j} \leftarrow Y_{i,j} + X_{i,k} \times W_{k,j}$;
  end
end
return Y

compute units. In the mean time, in order to reuse partial sums and avoid repeated data access, individual rows from the left (sparse) input matrix can be scheduled sequentially on the same compute units.

This work targets the larger FPGAs with thousands of DSPs available, which can be configured into more compute units than typical column dimension in the right input matrix. In order to fully utilize the compute availability, we must process multiple elements in the outer loop in parallel. Since elements of the same row must be accumulated on the same compute unit, we can only assign different rows spatially across compute units. This requires simultaneous data access for non-zero elements in multiple rows. Unfortunately, this is very difficult to achieve in standard sparse matrix formats, as shown in the next chapter. To make matters worse, DRAM typically requires serial access in order to maximize bandwidth, and it would be difficult to read data from arbitrary positions. In order to resolve this problem, a novel sparse matrix format is proposed in the next chapter.

3 PCOO Format

This chapter will first explore the existing sparse matrix formats, before proposing the novel packet-level column-only coordinate list format and the algorithm for its generation. This chapter will also describe the potential bank conflicts faced during computation, as well as its resolution during PCOO compression.

3.1 Sparse Matrix Formats

Table 1 shows that the adjacency matrix A is typically sparse. It is therefore crucial to represent it as some compressed format in order to avoid storing the >99% zeros. A few common sparse matrix formats include compressed sparse rows (CSR), compressed sparse columns (CSC), and coordinate list (COO).

The CSR and CSC formats first flatten the entire sparse matrix into row-major / column-major vectors, then they include a row / column pointer vector to store the start of each row / column as a position on the flattened vector. For an adjacency matrix of D-bit numbers with N rows, N columns, and M non-zero elements, CSR / CSC require MD bits to store the values, $M\log(N)$ bits to store the rows / columns, and $N\log(M)$ bits to store the row / column pointers. On the other hand, the COO format stores a row index, a column index, and the actual value. This requires a total of $2M\log(N) + MD$ bits.

It would be evident that these standard formats achieve very high compression rates, as they store little redundant information. However, the problem with these formats is that data access become much more complicated. In an uncompressed matrix, reading an element at row r and column c would simply involve calculating the position of the row and the offset of the element, only one data access for the value is required. In a CSR format, the processor must first read the row index, and then use the row index to compute the offset, before it is

finally able to use the offset to access the value.

Single data access is difficult, vectorized access is even more complicated. As mentioned in the previous chapter, in order to fully utilize the compute power of a large FPGA, multiple rows from the sparse matrix must be processed in parallel. This involves a vectorized read operation, and is most likely performed on the DRAM as smaller caches would not be able to store the large adjacency matrices. Due to the nature of DRAM access, sequential access can be done at approximately II=1, but it requires many cycles to locate an address for non-sequential access, therefore it is important that the access direction in the processing elements (PEs) match the storage direction in the DRAM. None of this is offered by existing formats, which motivates a new format.

3.2 Packet-Level Column-Only Coordinate List (PCOO)

The new format to be proposed is termed "packet-level column-only coordinate list (PCOO)", and is designed to support required DRAM access patterns while maintaining a low storage cost.

Each non-zero element in a vanilla COO format would contain a row index, a column index, and a value. However, the row information can be kept implicit and tracked within PEs. This way, only column information and element value require storage. The PE only needs to be informed of row starts and ends, when it would need to increment the internal row counter. This can be achieved via simple bitwise "headers". The PCOO format uses a three-bit header per element: start-of-row (SOR), end-of-row (EOR), and valid (VLD). The SOR bit indicates whether an element is the first non-zero of its row, the EOR bit indicates whether an element is the last non-zero of its row, and the VLD bit indicates whether an element should participate in computation or only in index tracking.

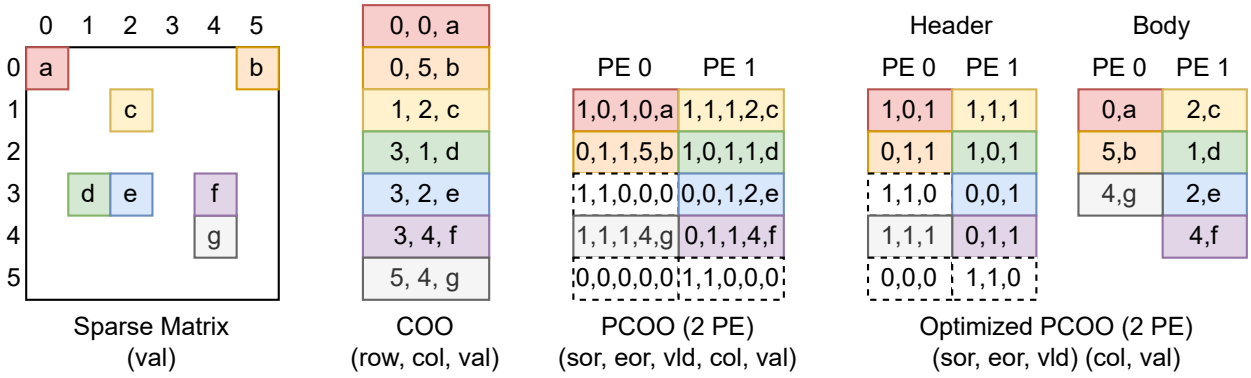

Figure 1: PCOO data format

In order to fully utilize the PE availability, multiple rows must be accessed in parallel. In order to fully utilize the DRAM bandwidth, data access must be sequential. In order to satisfy both these constraints, rows from the sparse input matrix are assigned to the PEs in a round-robin scheme. Given P PEs, row r in the sparse input matrix would be assigned to PE r mod P. Since the number of non-zero elements in the sparse matrix generally does not correlate with the row index, this assignment ensures an identical distribution of non-zero elements per PE, and consequently avoids idleness. In the rare case where the number of non-zero elements does correlate with the row indices, the node indices can simply be shuffled randomly. Since the graphs are unordered, this naturally would not affect the GCN results in any way. It would be shown in later chapters that the PE idle times are roughly the same in hundreds of randomly selected subgraphs. While it is possible for a more intelligent row assignment scheme to further maximize PE utilization, these methods would likely require larger preprocessing effort, and the marginal gains would be insignificant.

Now that each row is assigned to a given PE, non-zero elements in each row are packed into a "packet", and packets are concatenated sequentially for each PE. The workload for all PEs are interleaved to allow vectorized data access. Each packet is extended to match the length of the longest packet to preserve structural property. Figure 1 shows a PCOO compression example with a toy sparse matrix of 7 non-zero (colored) elements a through g. The solid borders in PCOO represent actual elements to participate in computation, while

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

Table 2: Compressed matrix sizes across datasets and algorithms
(Data from random-walk sampled subgraphs in GraphSAINT)

dashed-border elements with VLD=0 indicate injected elements as either empty rows or filler elements to maintain length.

Finally, the PCOO format is optimized to eliminate empty columns and values, as they consume the majority of data volume. As shown in "optimized PCOO" of Figure 1, only valid columns and values in the packets are preserved. This optimization significantly reduces redundant storage cost. Table 3.2 presents the average subgraph in the three datasets with its node and edge counts, as well as compressed format size. On average, this optimization preserves 71.2% of storage space that would have been expended for empty elements. Most of these empty elements are due to storing empty rows, as the adjacency matrix must be tiled to fit on-chip. Many nodes come with a degree of 1 or 2, resulting in many empty rows in each tile, and many empty injected elements with SOR=EOR=1 and VLD=0 to advance row pointers without affecting result.

This compression can be achieved in a simple algorithm with time complexity linear to number of non-zero elements in the input sparse matrix, as presented in Algorithm 3. Ideally, the resulting format is now ready to be easily decompressed on-chip and streamed into computation. However, realistically, there is still one preprocessing step to complete beforehand.

Algorithm 3: Sparse matrix preprocessing

```
inputs: X ∈ ℝ^{m×n}, T, K;
tiles, sor, eor, vld = [], T × 4, T × 2, T;
for t ← 0 to n − 1 by T do
    rows ← [[] for 0:K];
    for i ← 0:m do
        row ← [];
        for j ← t:(t + T − 1) do
            if X_{i,j} ≠ 0 then
                row.append(j % T + vld);
            end
        end
        row ← [0] if row is empty else row;
        row[0] ← row[0] + sor;
        row[-1] ← row[-1] + eor;
        rows[i % K].extend(row);
    end
    fill zeros until rows is rectangular;
    tiles.append(rows.transpose());
end
return tiles;
```

3.3 Bank Conflict

After data is loaded from off-chip DRAM, it is typically cached in on-chip SRAM. While the SRAM bandwidth and latency are much wider and shorter than DRAM, it still has its own limitations. Data in SRAM is stored in multiple "banks", each of which contains many addresses (i.e. 16 for LUTRAM, 512 for BRAM, 4096 for URAM, 3 common versions of on-chip caches on Xilinx FPGAs) and provides 1-2 ports. At any point in time, each port only allows access for one of the addresses, therefore it is usually unfeasible to access multiple addresses of the same bank in parallel. This phenomenon is known as "bank conflict". Due to the large data quantity in intermediate feature maps and weights, it is often necessary to fully utilize all available addresses.

For dense MM, data access is usually structured and sequential, and it is possible to layout data in a way that only a single address is required at a time. Unfortunately, such structural

Algorithm 4: Collision detection

inputs: $tile \in \mathbb{R}^{N \times K}$, depth d=16;
used, row ← [0 for 0:K], [-1 for 0:K];
result, share, block, j ← [], {}, {}, 0;
while *sum(used)* $< N \times T$ **do**
    i ← used$_j$;
    if $tile_{i,j} \in$ *share* ***or*** $tile_{i,j}$ % $d \notin$ *block* **then**
        row$_j$ ← $tile_{i,j}$;
        share.append($tile_{i,j}$);
        block.append($tile_{i,j}$ % d);
        used$_j$ ← used$_j$ + 1;
    else
        row$_j$ ← 0;
    end
    if *min(row)* ≠ *-1* **then**
        result.append(row);
        row, share, block ← [-1 for 0:K], {}, {};
    end
    j ← (j + 1) % K;
end
fill zeros until result is rectangular;
return result;

property is not available for SpMM. Each non-zero element in the left sparse matrix would map to a row on the right dense matrix. Since the column position of the sparse element cannot be known in advance, it could demand any corresponding row from the dense matrix. In order to process multiple elements from the sparse matrix in parallel, multiple rows from the dense matrix must be accessed in parallel, resulting in frequent bank conflicts. Naively accessing cached data without taking bank conflict into account would result in incorrect results. Furthermore, detecting conflicts during runtime and react accordingly would result in great complication on hardware.

In order to eliminate bank conflict, several design choices are made. First, since weight data are relatively small compared to feature maps, they can typically fit in the small LUTRAMs, which come with low capacity but also lower address size and higher bandwidth. The smaller address size and higher bandwidth allows mapping data horizontally across dif-

ferent banks instead of vertically across different addresses on the same bank, this drastically reduce bank conflict. Profiling data shows that the LUTRAM capacity is typically enough to store multiple copies of weight matrices on-chip, this opens the possibility to replicate the weight matrix. This way, even when accessing data that would have been on the same address, it is possible to route to a different replica of the weight matrix on a different bank, resolving the bank conflict problem. Finally, a collision detection algorithm is executed after compression to fully eliminate remaining bank conflicts, as presented in Algorithm 4. Empty elements are simply injected wherever there would be a bank conflict. Similar to the initial compression algorithm, the collision detection algorithm also comes with a time complexity linear to the number of non-zero elements in the sparse matrix.

The full preprocessing stage is implemented in C++, evaluation shows that the latency of all preprocessing is similar to the time required to read data from file. At this point, the adjacency matrix is ready to be processed on the hardware architecture, to be described in the next chapter.

4 Accelerator Architecture

In this chapter, the proposed accelerator architecture is discuss in detail, which efficiently supports the training process of quantized GCN.

4.1 Quantization

Similar to many other deep learning algorithms, GCN is equipped with multiple thousand parameters, a number generally more than enough to achieve maximum accuracy. The excess in computation opens an opportunity for simplification, an easy method is quantization.

The industry standard number representation for machine learning is 32-bit floating point numbers (FP32), a numerical format capable of representing up to approximately 10^{40} in range and down to 10^{-40} in precision. However, in practice, most learned parameters tend to center around zero with a standard deviation within 1 order of magnitude to 1. As a result, a large portion of the representable space of FP32 is wasted. In other words, it is often possible to represent the same numbers with smaller bit budgets without much loss of accuracy. In addition to storage costs, using high precision also demands larger computation units, which in turn consumes more DSPs to perform the same computation. In short, it is more favourable to use smaller representations, from 32 bit to 16 bit, and from floating point to fixed point.

In contrary from floating point numbers, fixed point numbers are simply integers, and do not carry an exponent portion in their representation. Values in each matrix share an implicit exponent, which allows integers to represent decimals to achieve higher precision. The advantage of fixed point is that there is no need to perform two separate computation for the floating point's exponent segment and the mantissa segment, which significantly reduces both area and energy cost per multiplier or accumulator. Naturally, the reduced representa-

tion comes at a cost in computation accuracy, which limits its usage. In GCN computation, experiments show that it is possible to quantize a majority of computation to 16-bit fixed point without any change in the algorithm, at an insignificant accuracy loss of within 1% and no change in training convergence time. There remain two exceptions which must be computed in FP32, including softmax and Adam update. Each of these operations require either a square, square root, or exponent operation, all of which unfortunately demand high precision. Detailed impact of quantization will be presented in the next chapter.

Quantization is performed via a simple algorithm. First, a fraction length F is set for each input matrix. Next, each value x is raised to $x(2^F)$ and rounded to the nearest integer. Finally, each value x outside the representable range of 16-bit signed integer (SINT16) is clipped to the nearest representable extreme (i.e. -2^{15} or $2^{15}-1$). This results in a matrix of integers, which can now be type cast to SINT16. The fraction length F is determined through trial and error in a toy iteration. For instance, the training algorithm is first run in full precision for a single epoch on a single subgraph. Each feature map and weight matrix is then quantized with every choice of F between -16 and 32. The quantization attempt per F choice is then compared against the FP32 version to compute a mean squared error (MSE), finally the F choice that yields the lowest MSE is assigned to the matrix. Similar quantization methods have been previously proposed to improve GCN efficiency [11], and have achieved even lower representations. However, they require changes to the network structure, and is therefore less scalable than a native approach.

4.2 Overall Architecture

The GCN workload is split between the FPGA device and the CPU host. The MM and SpMM in forward and backward propagation are quantized to SINT16 and computed on the FPGA. The element-wise additions and multiplications are also performed on the FPGA as

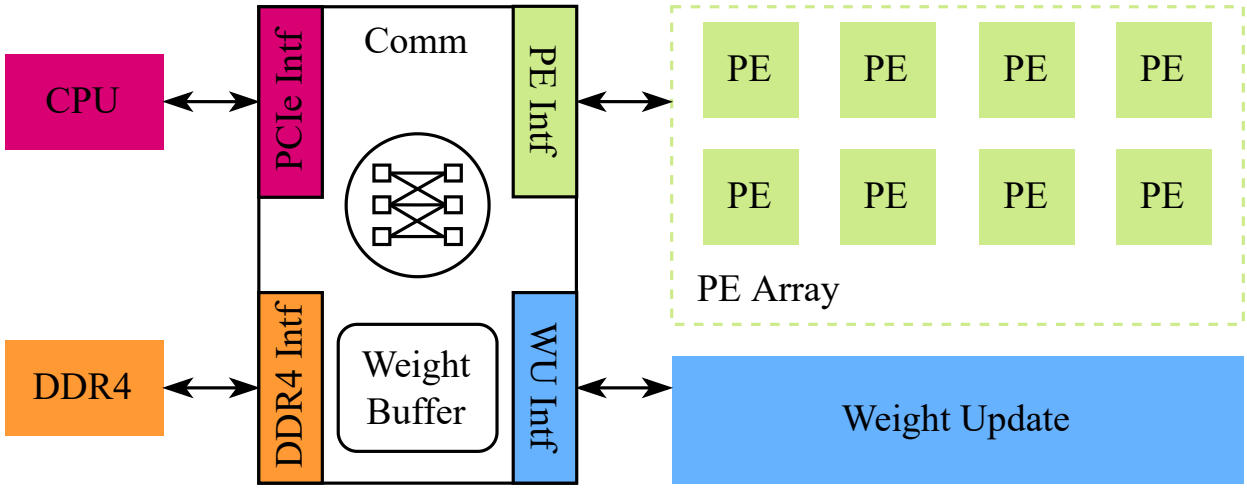

Figure 2: Overall architecture of **SkeletonGCN**.

soon as the gradients are computed during the backward phase. On the other hand, since softmax and the square / square-root portion of weight update require hardware expensive units to compute accurately, they are assigned to the CPU. Other software processes, such as graph sampling, data preprocessing, are also assigned to the CPU. The detailed scheduling between CPU and FPGA will be discussed in Section 4.6.

Regarding workload assignment, the overall architecture is shown in Figure 2. The *PE Array Module* performs all the operations in the forward and backward phases, while the *Weight Update Module* subsequently updates the weights after gradients are obtained from back-propagation. The *Communication Module* is responsible for the data transfer between CPU and FPGA, between off-chip memory (DDR4) and FPGA, and also among different PEs.

4.3 Unified PE Architecture

The performance of the training accelerator comes from both reducing the off-chip memory access overhead, as well as increasing the utilization of computation resources. Since the full input graph is sampled, and only one subgraph is trained at every point in time, it is

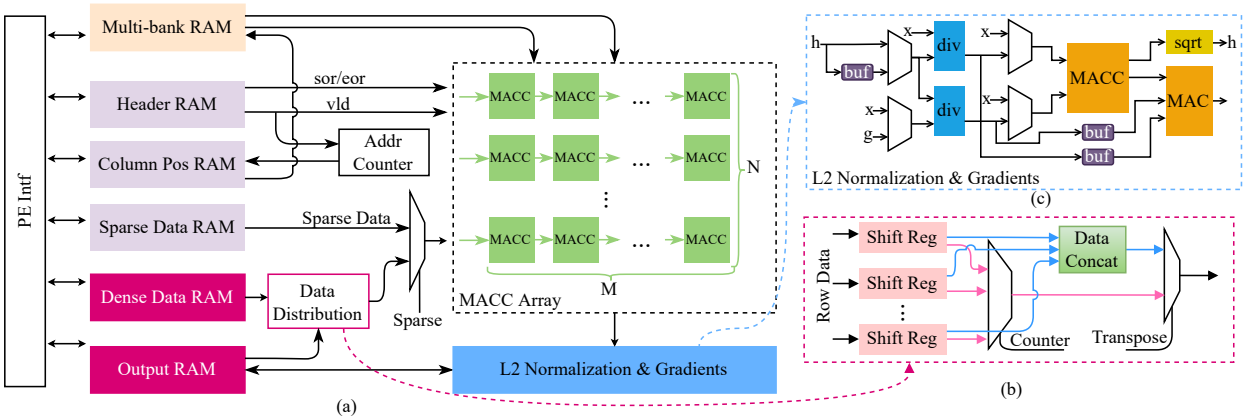

Figure 3: (a) The architecture of the unified PE. (b) The architecture of the L2 normalization and gradients module. (c) The architecture of the data distribution module.

possible to set the size of the subgraph appropriate so that all the data of a single subgraph can be cached in the on-chip memory availability of the FPGA. This way, the FPGA only requires access from the off-chip memory three times during training per subgraph. The first access is to obtain the initial data for the forward pass. The second access is to send back the results of the last layer to compute loss and gradients on CPU. Finally, the gradients of the last layer for back-propagation must be streamed back to the FPGA to carry out the remainder of back propagation.

To increase computation efficiency, a unified PE architecture is designed to perform each step in each layer during the forward and backward phases, as shown in Figure 3 (a). As previously mentioned, the main operations during forward and backward are SpMM, MM and TMM, where the majority of basic operations are multiply-accumulate. In order to support these operations, the accelerator is designed with an $M \times N$ *Multiply-Accumulator (MACC) Array* to spatially expand each MM version. Generally, a row from the right dense matrix is routed to a row of MACC units, while a single element from the left sparse matrix is multi-cast across said row. The challenge then becomes how to feed data into the MACCs to maximize their efficiency under different workloads.

SpMM

As discussed in previous chapters, the optimized PCOO format is used to compress sparse matrices, before storing the compressed data (header, body and non-zero elements) into 3 separate RAMs, as shown in Figure 3 (a). During computation, the Header (SOR/EOR/VLD) is flushed out from the *Header RAM*, when the "VLD" signal is used to enable the *Address Counter*, which is in turn used to generate the address of the *Body RAM*. The output of the *Body RAM* is the column position of the non-zero element in the sparse matrix, which indicates the address of the corresponding dense data in the *Multi-bank RAM*. This simplifies the decompression logic of the sparse matrices under the optimized PCOO format, and can be implemented with several wires and a counter, as shown in Figure 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 remain active during most cycles of SpMM computation, thus leading to a high DSP efficiency. Evaluation details will be discussed in the next chapter.

4.3.1 MM and TMM

Although MM and TMM share the same computation operations, they require different memory access patterns for ordinary matrices and transposed matrices. Moreover, the output of one layer can be either used in standard arrangement or the transposed arrangement for subsequent computations. To alleviate memory access burden and improve DSP efficiency, the same memory load and store logic is performed 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 Figure 3 (b). The row data of the left matrix in MM or TMM is first fetched from the on-chip memory, before it is fed into the shift registers.

Each shift register stores a single row and can be configured to output either one element or all the elements in the row, according to the computation mode. When computing MM, each shift register is first configured to output one element. Next, all the elements from different shift registers are concatenated and fed into the N rows of the *MACC Array*, as shown with the blue arrow in Figure 3 (b).

On the other hand, when computing TMM, each shift register is first configured to output all the elements in one row. Then, the elements are selected one by one to feed into the N rows of the *MACC Array*, as shown with the pink arrow in Figure 3 (b). By setting the data width of the on-chip memory as $N \times N \times 16$ (for 16-bit signed integers), active data can be streamed out every cycle for both MM and TMM to ensure the MACCs remain active, consequently maintains high DSP utilization. Since the MACC row size $N = 16$ is set in the accelerator, the data width of $N \times N \times 16$ is easy to achieve by using block RAMs (BRAMs) or ultra RAMs (URAMs) in Xilinx FPGA.

4.3.2 L2 normalization and its gradients

The L2 normalization operation and its gradient computation follow a MM in the forward and backward phase respectively. Therefore, an extra module is designated to receive the results of the *MACC Array* as inputs to pipeline the computation. The CORDIC IP and division IP in Xilinx FPGA are used to compute square root and division needed by L2 normalization, respectively. As the L2 normalization and its gradients are computed in serial, most of the computation units are reused to minimize resource utilization, as shown in Figure 3 (c). All the multiplexers in Figure 3 (c) are selected by the signal indicating the computation of L2 normalization or gradients. As expressed in Equation 3 and Equation 10, the term $||X_{l-1}||_2$, which is produced by the square root module, is used in both computing L2 normalization and its gradients. Therefore, the results of the square root module h are buffered to eliminate redundant computation. Moreover, $\frac{\frac{\partial \mathcal{L}}{\partial X_{l-1}}}{||X_{l-1}||_2}$ and $\frac{X_{l-1}}{||X_{l-1}||_2}$ are also used

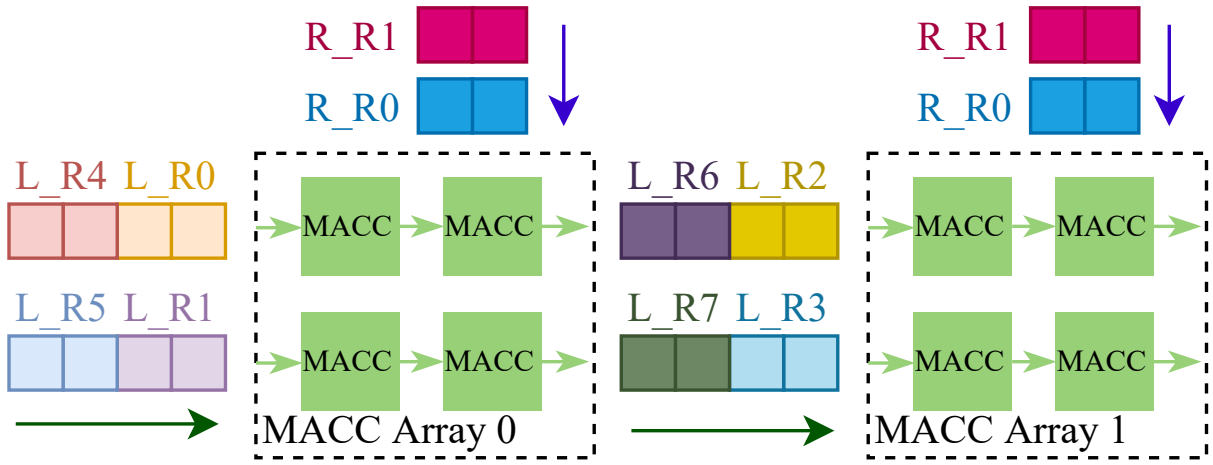

Figure 4: An example of allocating SpMM, MM or TMM onto 2 *MACC Arrays*. The notion "L_R#" indicates the row index of the left matrix while the notion "R_R#" indicates the row index of the right matrix.

several times in different computation steps for computing the gradients. Therefore, they are buffered after the first computation and reuse them to save computation resources and reduce latency.

4.4 Weight Update

Since the weights are updated after computing the gradients by TMM, a separate module is designed to buffer the outputs of the PE array to fully pipeline the computation, as shown in Figure 2. The square root and division operations in Adam are also computed by using CORDIC and Division IPs in Xilinx FPGA. Moreover, the hyper parameters in Adam are to the nearest power of 2 (*i.e.*, learning rate η) to simplify the multiplications to a bitshift.

4.5 Data Communication

The *Communication Module* handles data transfer between external memory and FPGA (both with CPU and DDR4), and among different PEs, as shown in Figure 2. The PCIe

Gen3 X16 interface used for communication between CPU and FPGA, and only the initial data, final results of the forward phase and the first gradients for the backward phase are transferred via PCIe. DDR4 is also used as the external memory to save initial data for different training epochs and on-chip buffers are designed to perform 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 to perform in parallel on different PEs (see allocation details in Section 4.6), data is also transferred among neighbor PEs. Result gradient matrices from each weight matrix are streamed to 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), FIFOs are used in the *PE Interface Modules* to control the bandwidth between different PEs.

4.6 Allocation and Scheduling

This section will discuss the allocation of SpMM, MM and TMM onto different PEs, as well as the computation scheduling between CPU and individual FPGA modules.

4.6.1 Allocation

For SpMM, MM and TMM, a round robin method is used to assign different rows of the left matrix onto different rows of different PEs, as shown with a simple example in Figure 4. This hides the row information of the non-zero elements in the sparse matrix under the row index of the *MACC Array* in each PE, thus reducing the complexity and memory requirements of the optimized PCOO format. Moreover, the element of each row in the left matrix is fed into the MACCs one by one, and the element is accumulated to get the MM results. Therefore, the elements of the right matrix are fed into the PEs row by row, as shown by the blue arrow in Figure 4. Since it is beyond the capacity availability to perform computation

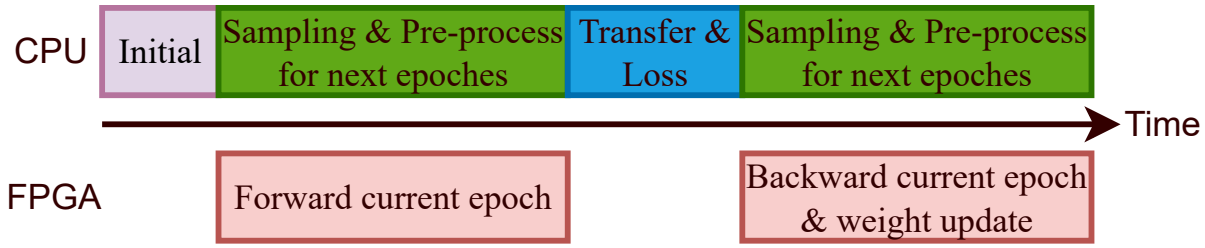

Figure 5: Scheduling between CPU and FPGA.

of the full matrix, a row-wise partition in the left matrix and a column-wise partition in the right matrix is performed to fit a single tile into the *MACC Array* of each PE. This way, each PE can only receive a portion rows of the result matrix (in the example shown in Figure 4, *MACC Array 0* receives rows 0/1/4/5 of the result matrix while *MACC Array 1* receives rows 2/3/6/7). Since each *MACC Array* only uses a portion of the rows in the left matrix, the results are propagated within each PE if the result matrix is used as an input in subsequent operations. Otherwise, they will be communicated among different PEs to collect the full result matrix.

4.6.2 Scheduling

During the training of GCN, most of the computation expensive operations are assigned to the FPGA while others are assigned to CPU. In order to improve the overall training performance, most of the operations between CPU and FPGA are executed in parallel, as shown in Figure 5. After the first data initialization, the CPU repeatedly sample and preprocess data for the next epochs while the FPGA performs forward propagation in the current epoch / subgraph. The CPU is interrupted to transfer forward results and compute the softmax, cross-entropy loss and the corresponding gradients once the forward phase of one epoch is finished on the FPGA. The gradients are then transferred back to the FPGA for backward and the CPU is back to preparing data for next epochs. Once the data of one epoch is prepared on the CPU, data will be transferred to the FPGA via PCIe and saved

to the DDR4 for later use. In addition, the multi-core CPU can be set to work in parallel because the subgraphs are independent to each other.

5 Evaluation

In this chapter, the proposed approach is evaluated with comprehensive experiments. The proposed 16-bit signed integer training process is first evaluated on different datasets and networks to show its effectiveness. The FPGA accelerator is then evaluated on GraphSAINT with different configurations. Finally, a comparison against the state-of-the-art FPGA accelerator with the same FPGA configurations is made to show overall effectiveness.

5.1 Experimental Setup

The accelerator design is implemented with Verilog HDL and deployed on a Xilinx Alveo U200 board. In order to fully utilize the resource availability of the U200 board, eight processing elements (PEs) are implemented, each PE is equipped with a 32×16 *MACC Array* for SpMM, MM and TMM. The BRAMs and URAMs of the U200 board are used to store all the data used. After synthesis and implementation with Vivado 2020.1, the overall resource utilization is shown in Table 5.2. The accelerator relies heavily on on-chip memory (LUTRAM, BRAM, and URAM) to buffer the graph data and thus satisfies bandwidth constraints.

As shown in Table 1, three common graph datasets are used for evaluation. Since the graph datasets are too large to buffer on board, the same sampling algorithms as GraphSAINT and GraphACT are used for fair comparison. All the results on CPU and GPU are generated by using PyTorch Geometric and the open-sourced codes provided by GraphSAINT and GraphACT.

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

Table 3: Resource Utilization on Alveo U200 Board.

5.2 Training Accuracy and Latency

Figure 6 shows a training accuracy cross-comparison between three GraphSAINT implementations across three popular datasets. The original GraphSAINT configuration uses large subgraphs (*i.e.,* 8000 nodes) as proposed by the original authors. The simplified version uses 2000 node subgraphs to better fit the FPGA, and removes a portion of functionality including dropout and batch normalization. The quantized version is executed on this accelerator, mostly in 16-bit signed integer as mentioned in previous sections. Experiment results show insignificant drops of approximately 0.5-0.7% in F1 score for Reddit and Yelp datasets. On the other hand, the drop for PPI was significant at 8% because the PPI dataset is less robust to smaller subgraph sizes during sampling, this was discussed in the original GraphSAINT text, and this accuracy reduction would be eliminated with larger subgraph sizes (i.e. 4000 nodes).

The training latency under the same configurations is shown in Table 4. On average, this accelerator achieves 53.5× and 1.7× speedup compared with the state-of-the-art PyTorch Geometric implementation, executed on the Intel Xeon CPU and Nvidia Tesla P100 GPU, respectively.

5.3 Comparison with State-of-the-art

Training latency is also compared with GraphACT to further show the effectiveness of our approach. For a fair comparison, the experimental settings, including network architecture,

Table 4: Comparison with GraphACT, CPU, GPU on GCN and GraphSAINT.
"-" indicates no reported results.
"*" indicates results directly taken from GraphACT

		GraphACT	CPU	GPU	Proposed
Data type		Float32	Float32	Float32	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

testing datasets and FPGA board are all set to match GraphACT reporting. The GCN evaluated has two graph convolution layers and one MLP layer in the classifier, and the hidden size is set to 256 for all graph convolution layers. As shown in Table 4, this accelerator achieves speedup up to 11.3× compared with GraphACT across all datasets. On average, 8.7× speedup is achieved across PPI, Reddit, and Yelp datasets.

The advantages come from both the quantization-aware training algorithm, as well as the unified PE architecture. Firstly, as mentioned in previous sections, precision is reduced from 32-bit floating point to 16-bit signed integers with negligible accuracy loss, which in turn greatly reduces the usage of DSPs (in Xilinx FPGA, each Float32 multiplier consumes three DSPs while each INT16 multiplier-accumulator only consumes one DSP). Therefore, more multipliers than GraphACT for computation with same number of DSPs. Secondly, the design choice of a unified PE architecture dramatically increases the DSP efficiency. In GraphACT, separate modules are used for feature aggregation and weight transformation. Although its scheduling algorithm tries to overlap the operations of both modules, there are still quantitative idle cycles for either of the two modules.

5.4 Discussion

This section will discuss the DSP efficiency, which dramatically influence the overall training latency. The DSP efficiency is defined as follows:

$$DSP\_EFF = \frac{Lat_{theo}}{Lat_{test}}, \tag{12}$$

where Lat_{theo} and Lat_{test} indicate the theoretical latency and tested latency, respectively. All the zeros in SpMM and the theoretical latency of SpMM and MM are then calculated by using Equation 13 and 14.

$$Lat_{theo}^{SpMM} = \frac{\#\ of\ non-zero\ MAC\ ops}{\#\ of\ MAC\ units}. \tag{13}$$

$$Lat_{theo}^{MM} = \frac{\#\ of\ MAC\ ops}{\#\ of\ MAC\ units}. \tag{14}$$

Following the above definitions, the average DSP efficiency of training GCN on the Reddit dataset is analyzed. As shown in Figure 7, the DSP efficiency for computing MM and TMM can be as high as 98.3% for some cases. On the other hand, the DSP efficiency of SpMM is only 71.2% because empty elements are injected to avoid bank conflicts, as mentioned in PCOO compression. However, it has little influence on the total training latency because SpMM only accounts for approximately 1% of the total computation workloads.

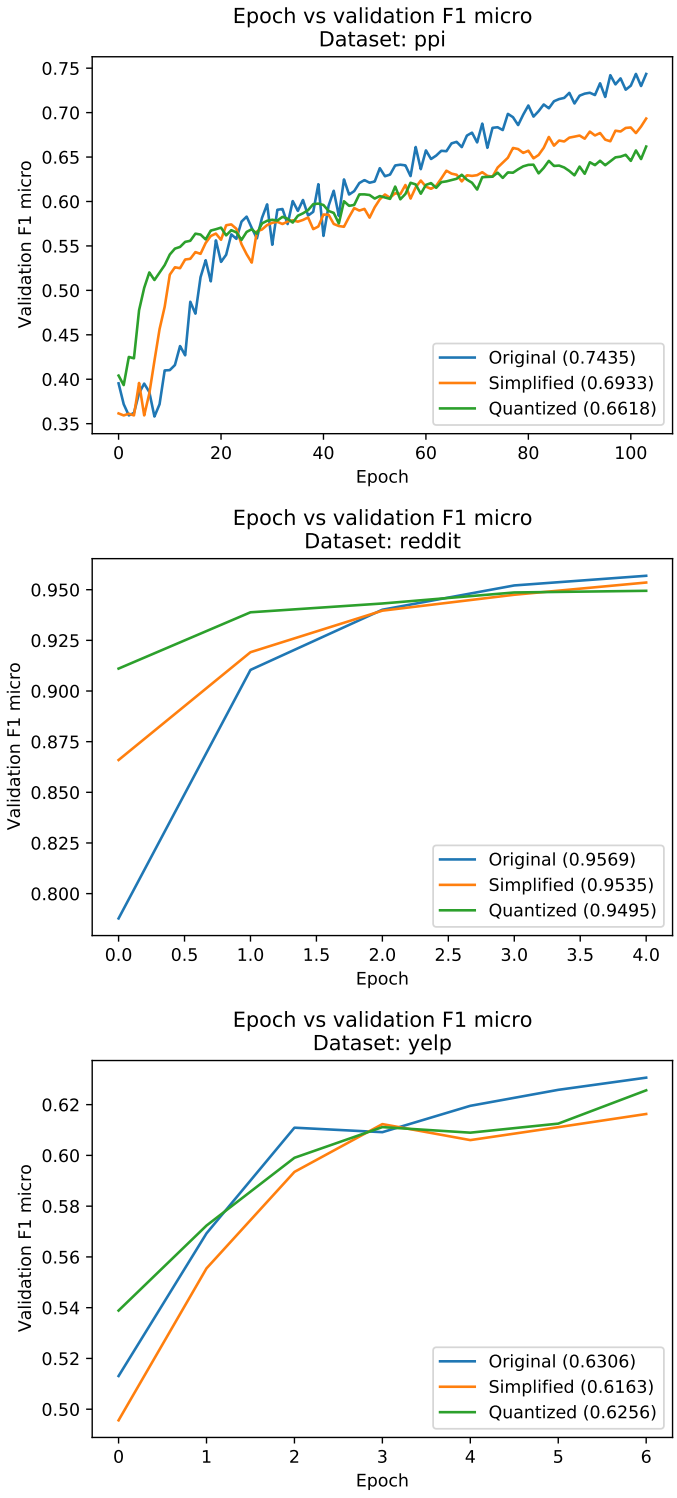

Figure 6: Training Accuracy Comparison

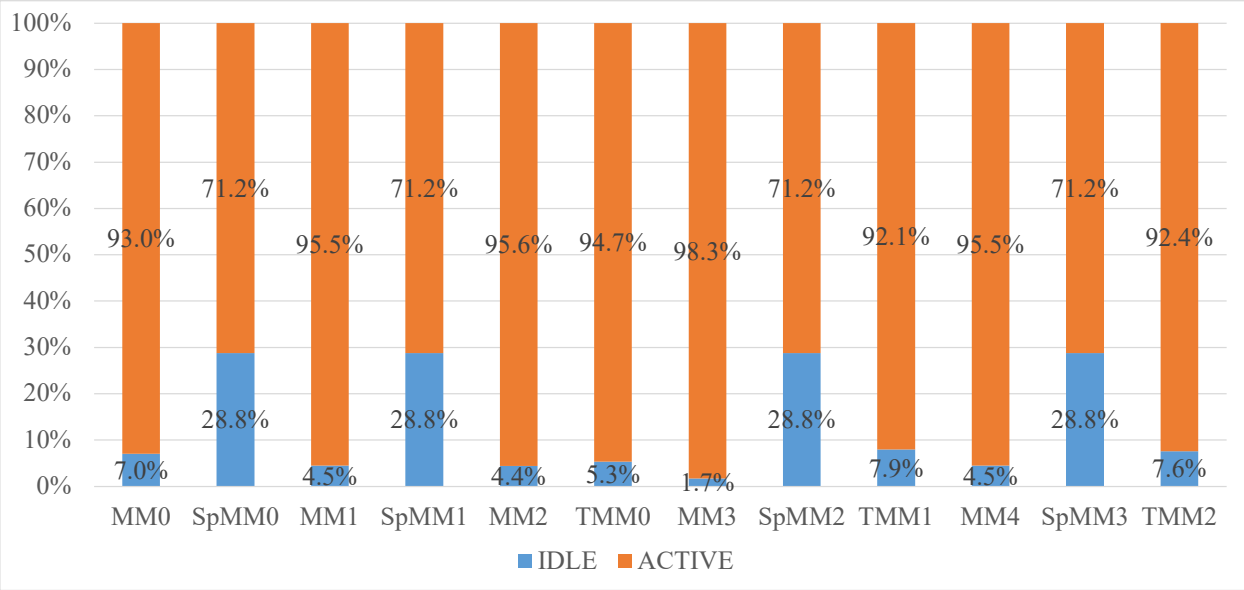

Figure 7: The DSP efficiency of computing SpMM, MM, and TMM in training GCN on Reddit

6 Conclusion

In this work, a software-hardware co-optimized GCN accelerator on FPGA is proposed in order to improve GCN training efficiency. The data representation graph inputs and trainable parameters are first quantized from 32-bit floating point to 16-bit signed integer to reduce computation and storage requirements. The non-linear operations are simplified and redundant computations are eliminated, in order to better fit the computation on FPGA. Next, a linear time sparse matrix compression algorithm is employed to further reduce memory bandwidth while enabling efficient decompression on hardware. A unified hardware architecture is then proposed to compute SpMM, MM and transposed MM to improve DSP efficiency. Finally, evaluation shows that the simplified training approach can train the network with negligible accuracy loss. In terms of efficiency, the new accelerator achieves up to 11.3× speedup over existing FPGA-based accelerator while executing the same network structure and maintaining the same training accuracy. It also achieves up to 178× and 13.1× speedup over state-of-art CPU and GPU implementation, respectively.

References

[1] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve 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)*, pages 922–936. IEEE, 2020.

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

[3] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

[5] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[6] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[7] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 775–788. IEEE, 2021.

[8] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Transactions on Computers*, 70(9):1511–1525, 2020.

[9] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[10] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[11] Shyam A Tailor, Javier Fernandez-Marques, and Nicholas D Lane. Degree-quant: Quantization-aware training for graph neural networks. *arXiv preprint arXiv:2008.05000*, 2020.

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

[13] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[14] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

[15] Hanqing Zeng and Viktor Prasanna. Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 255–265, 2020.

[16] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

[17] Xiaojin Zhu, Zoubin Ghahramani, and John D Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International conference on Machine learning (ICML-03)*, pages 912–919, 2003.