VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

COMPUTER ENGINEERING PROJECT

SEMESTER 241

ACCELERATION OF GRAPH ATTENTION NETWORK ON FPGA

MAJOR: COMPUTER ENGINEERING

COMMITTEE: CE - CC05

Supervisors:

Assoc. Prof. Dr. Pham Quoc Cuong - HCMUT

Authors:

Hoang Tien Duc - 2152520

Dang Hoang Gia - 2153312

Nguyen Duc Bao Huy - 2152089

HO CHI MINH CITY - December 2024

Instructor's Signature

_____ Date: _____

Assoc. Prof. Dr. Pham Quoc Cuong

Faculty of Computer Science and Engineering

Contents

List of Figures

List of Tables

Acknowledgement

We would like to express my appreciation to <NAME> for his invaluable guidance. Our research has greatly benefited from his deep knowledge, perceptive criticism, and constant support.

Besides, we are grateful for the help from ... for valuable comments for research aspect and our proposed pipeline, including pointing out strengths as well as weaknesses of previous works on multimodal models so that we can elaborate on our ideas more effectively.

In addition, we would like to extend our gratitude to my family. Their unwavering faith in our abilities and constant encouragement have been my pillars of strength. Their belief in my potential has been a constant source of motivation and resilience. We are forever grateful for their love and support.

Abstract

List of Abbreviations

Abbreviation	Definitions
SPMM	Sparse Matrix Multiplication
SP-PE	Sparse Processing Element
DMVM	Dense Matrix Vector Multiplication
GAT	Graph Attention Network
GCN	Graph Convolution Network
FPGA	Field-Programmable Gate Array
DMA	Direct Memory Access
CDMA	Central Direct Memory Access
AXI	Advanced eXtensible Interface
FIFO	First In First Out
LUT	Look-up Table
RAM	Random Memory Access
BRAM	Block RAM
DSP	Digital Signal Processing

Chapter 1

Introduction

1.1 Introduction

Graph Neural Networks (GNNs) have shown remarkable performance across various domains, such as networking, biology, and recommendation systems, by learning from graph-structured data. Graph Convolutional Networks (GCNs), inspired by Convolutional Neural Networks (CNNs), have been widely applied in tasks like node classification, link prediction, and graph classification. However, GCNs primarily focus on learning graph structure parameters, limiting their effectiveness in handling inductive tasks.

To address these limitations, the Graph Attention Network (GAT) introduces an attention mechanism, allowing for better adaptability and efficiency in handling inductive tasks. GATs outperform GCNs in various applications by overcoming the rigid and monolithic nature of GCNs, thus making them more suitable for real-time tasks and dynamic graph structures.

While CPUs are efficient for control-intensive tasks, they are not optimal for the parallel nature of GAT inference. Field Programmable Gate Logic, with its high parallelism and flexibility, is well-suited for accelerating GAT computations. It can efficiently handle parallel tasks, optimize for specific operations like attention and aggregation, and achieve high performance with lower power consumption, making it an ideal choice for large-scale, inductive GAT applications.

1.2 Research objective

The thesis is comprised of two main phases, with each phase focusing on specific objectives:

Phase 1: Computer Engineering Project

- Study Graph Attention Network and Related Applications: Conduct a detailed exploration of GAT, focusing on their theoretical foundations, compare with GNNs, and other existing challenges.
- Training Model: Choose a dataset and train model with several methods to adapt with hardware resource, such as quantization, finetuning,...
- Design of Acceleration Core for GAT: Develop the hardware accelerator architecture based on GAT computations, with a focus on optimizing operations like matrix multiplication or aggregation.
- Testing and Verification: Create test plan to test the main module to validate the design concept and functionality

Phase 2: Capstone Project

- Complete Simulation and Design Verification: Run simulations to verify the design's functionality, checking for correctness and efficiency.
- Design Optimization: Optimize the design to make it more flexible, improving performance and adaptability.
- Implementation and Testing on Xilinx FPGA: Integrate the design and run tests on the Xilinx FPGA platform, assessing real-world performance.
- Final Report: Write a comprehensive final report summarizing the research, design process, and experimental results.

1.3 Research scope

This thesis focuses on the integration of artificial intelligence (AI) and hardware systems, specifically in the implementation of machine learning and deep learning algorithms using hardware description languages. The research involves using key components such as the AMBA AXI protocol, DMA, and the Zynq Processing System on the Zynq MPSoC Ultrascale+ platform, which are supported by Xilinx. However, the scope does not include an in-depth analysis or detailed exploration of these components, as they are treated as supporting tools rather than core research areas.

The primary objective is to evaluate the performance of a hardware-accelerated model on FPGA, with the emphasis placed on metrics such as execution speed, accuracy, and system efficiency. The study concentrates on assessing the effectiveness of the hardware implementation rather than focusing on software-based or theoretical aspects of machine learning. By maintaining a clear focus on the performance evaluation of the FPGA-based accelerator, this research aims to deliver practical and measurable outcomes relevant to the field of hardware-accelerated AI systems.

1.4 Research subject

Research will focus on four main topics:

- Graph Attention Network: Mathematical model specification and its applications.
- Multiprocessing System on Chip: Data transfer mechanism from the Processing System (PS) to the Programmable Logic (PL).
- Hardware Architecture Specification: Design and implementation of the hardware architecture for the Graph Attention Network.
- System Evaluation: Technique to evaluate the performance of the hardware design.

1.5 Outline

The rest of the thesis is organized as follows:

- **Chapter 1 - Introduction:** Depicts an general scenario of the research
- **Chapter 2 - Background:** Discusses the foundation concepts, theories and technologies relevant to the thesis, such as Graph Theory, Graph Attention Network algorithms, Attention Mechanism. This section also includes the related works of FPGA-based GNNs and GATs.
- **Chapter 3 - Related Architecture:** Discuss the data preprocessing methods as well as modification in graph data format and GAT calculations
- **Chapter 4 - Overall Architecture:** top-down approach of the hardware architecture design and illustration of each block designs.
- **Chapter 5 - Implementation:** illustrates specific implementation of the system
- **Chapter 6 - Experiment results:** provides verification results, synthesis results of the implemented modules, reports the resource usage of the system
- **Chapter 7 - Future plan:** Discuses Improvements, Extensions and Project plan in the next semesters.
- **Chapter 8 - Conclusion:** Summarizes the findings, results from the researchs.

Chapter 2

Background

This chapter introduces key concepts needed for the research, including graph theory, Graph Convolutional Networks (GCNs), and their application in machine learning. It also covers FPGA acceleration, explaining how hardware optimization enhances graph-based computations. Additionally, the chapter introduces the FPGA board (ZCU106) and PYNQ framework used in the project, providing a foundation for the implementation.

2.1 Field Programmable Gate Array and System on Chip

A field-programmable gate array (FPGA) is a type of configurable integrated circuit, including its ability to be reprogrammed post-manufacturing to perform specific logical operations.

FPGAs contain configurable logic blocks (CLBs) and a set of programmable interconnects (ICs) that allow the designer to connect blocks and configure them to perform everything from simple logic gates to complex functions. Some main components in FPGA would be briefly described:

- CLB: these are the basic cells of FPGA. It consists of one 8-bit function generator, two 16-bit function generators, two registers (flip-flops or latches), and reprogrammable routing controls (multiplexers). The CLBs are applied to implement other designed functions. Each CLBs have inputs on each side which make them flexible for mapping and partitioning of logic.

- I/O Pads: The Input/Output pads used for the outside peripherals such as USB, JTAG, UART,... to access the functions of FPGA and by using the I/O pads, it can also communicate with FPGA for different applications using different peripherals.
- Interconnection wires: also called as switch matrix. it is used in FPGA to connect to the long and short interconnection wires together in flexible combination.

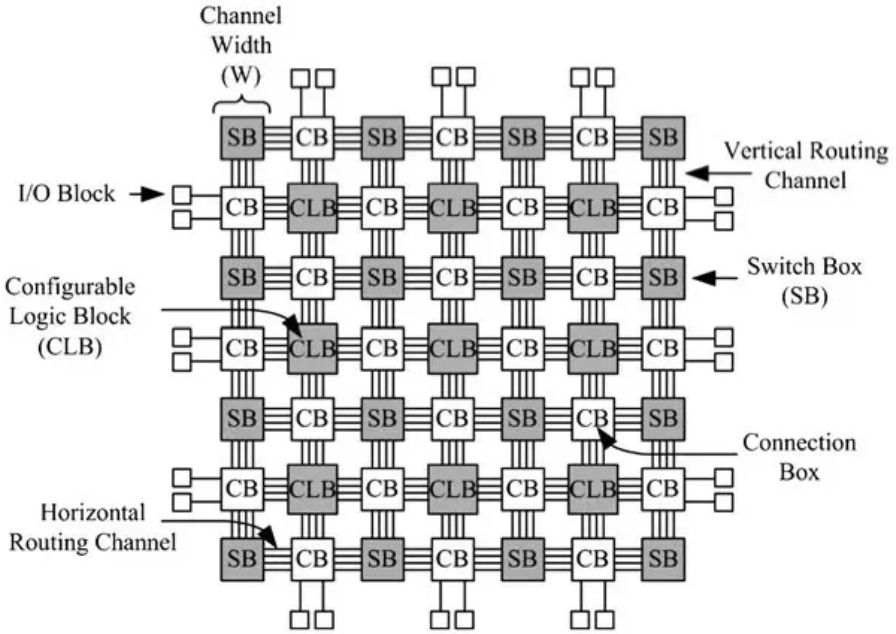


Figure 2.1: Components in FPGA

Due to its reconfigurable features, engineers and researchers choose to implement their work on FPGA by using Hardware Description Languages (HDLs). Nowadays, the well-known HDL and mostly be used are Verilog, System Verilog and VHDL. Some open-source organization also contruct a specified modern HDL, like Chisel of ChipAlliance. To those who are familiar with Verilog, SystemVerilog can be a good approach for modern logic design.

With the flexibility and parallel processing capabilities of FPGAs, designers (so-called RTL Design Engineer) are motivated to integrate them into System-On-Chip Architectures because of their ability to handle specific, compute-intensive tasks more efficiently than general-purpose processors. System-on-chip refers to an integrated circuit that comprises of hardware components of a computer or electronic system , for example DMA, CPU, Bus protocols,... onto a single chip. FPGAs can be integrated into SoC to handle complex programable tasks that CPU and other blocks in SoC cannot optimally

compute, such as image processing, machine learning inference or Network Security protocols. Additionally, The reprogrammable feature of FPGAs allows Designers to update and modify the logic, making it suitable for research and new discoveries in various fields, especially A.I Model and Digital Signal Processing. As a result, SoC helps to speed up data access and reduce power comsumption while FPGAs offer configurable logic for tailored computing tasks. This heterogeneous system combines the streamlined integration of an SoC with the flexible, programmable capabilities of an FPGA, optimizing both functionality and efficiency.

2.2 Zynq Ultrascale+ MPSoC ZCU106

A Multiprocessor System-on-Chip (MPSoC) is an integrated circuit that combines **multiple processor cores** and various system components **onto a single chip**. This integration enhances performance, reduces power consumption, and minimizes physical space requirements, making MPSoCs ideal for complex, high-performance applications.

The Zynq UltraScale+ MPSoC is a family of advanced System-on-Chip (SoC) devices developed by AMD (formerly Xilinx). These devices integrate multiple processing units, programmable logic, and various system components onto a single chip, offering a versatile platform for a wide range of applications.

The ZCU106 Evaluation Kit by Xilinx is a development platform designed for performance-intensive applications such as video conferencing, surveillance, ADAS, and streaming. It features a Zynq UltraScale+ MPSoC with a quad-core Arm Cortex-A53, dual-core Cortex-R5, Mali-400 MP2 GPU, and a Video Codec Unit supporting H.264/H.265 4K@60fps. High-speed interfaces include HDMI, PCIe Gen3x4, USB 3.0, DisplayPort, and 2x SFP+ cages, with programmable logic based on AMD's UltraScale architecture for hardware customization. Memory resources include 72-bit DDR4 with ECC for the processing system and 64-bit DDR4 for programmable logic. Development is supported by the Vivado Design Suite and PetaLinux, making it an efficient platform for debugging and validating hardware-accelerated designs.

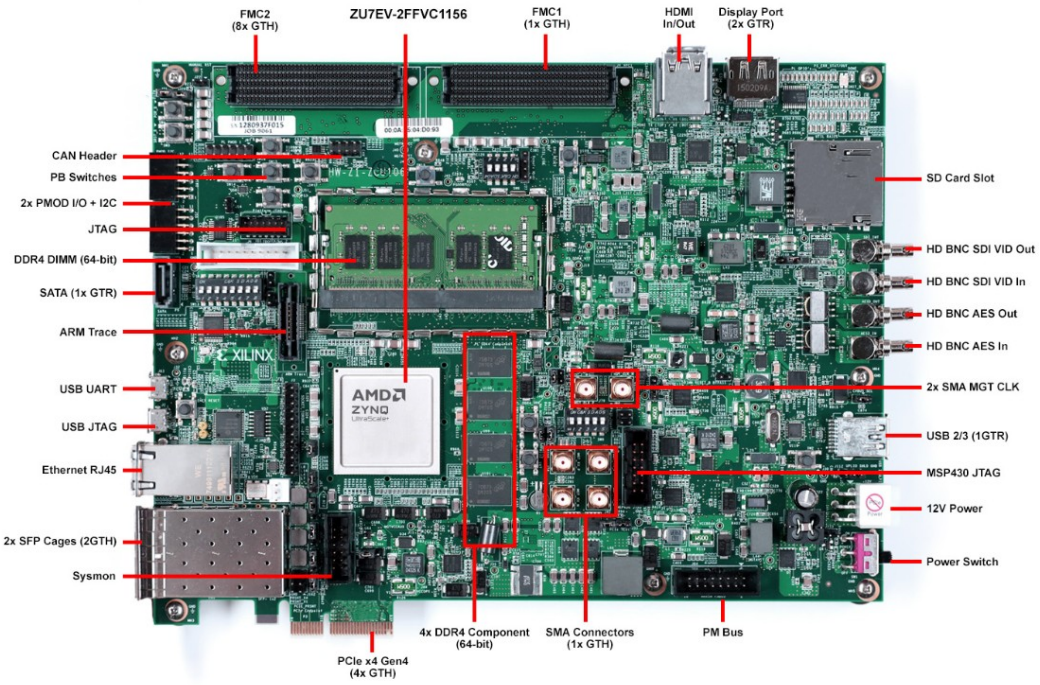


Figure 2.2: ZCU106

Resource	Information
Core Device	XCZU7EV
Available IOBs	260
LUT Elements	230,400
Flip-Flops	460,800
Block RAMs	312
Ultra RAMs	96
DSPs	1728

Table 2.1: Resource on XCZU7EV

2.3 PYNQ Framework

Python Productivity for Zynq (PYNQ) is an open-source framework developed by AMD (formerly Xilinx) to simplify programming on Zynq and Zynq Ultrascale+ MPSoC devices. Software developers as well as hardware designers need to control the programmable logic (FPGA) and processing system (CPU) using Python. By providing Jupyter notebooks and the huge ecosystem of Python libraries, PYNQ allows designers to combine

specific tasks from software such as Data visualization, and AI modeling with hardware works like signal processing or AI accelerator. Hence, PYNQ's flexibility and ease of use make it a popular choice for prototyping, research, and educational purposes.

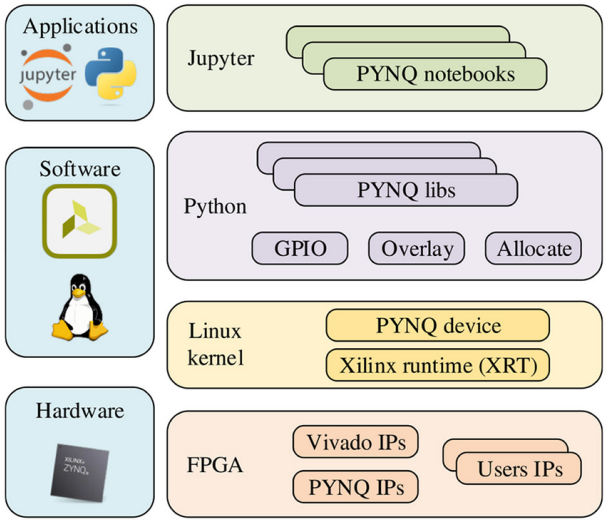


Figure 2.3: Pynq Framework

2.4 Graph Introduction

2.4.1 Graph Theory Overview

A graph is a data structure consisting of a set of vertices and edges used to represent connections. In other words, A graph represents the relations (edges) between a collection of entities (nodes). Graph features help describe complex relationships in various fields, such as path findings, social network analysis, economic business analysis,...

2.4.2 Basic components of a Graph

A graph comprises of three components: Vertices (Nodes), Egdes, Weights (optional).

- Vertices (V): Represent entities in the model
- Egdes (E): Illustrate the connections or interactions between vertices. Edges could be directed or undirected.
- Weights (W): Edges can have weights, representing cost, distance, correlation or any parameter that measure the connection of nodes.

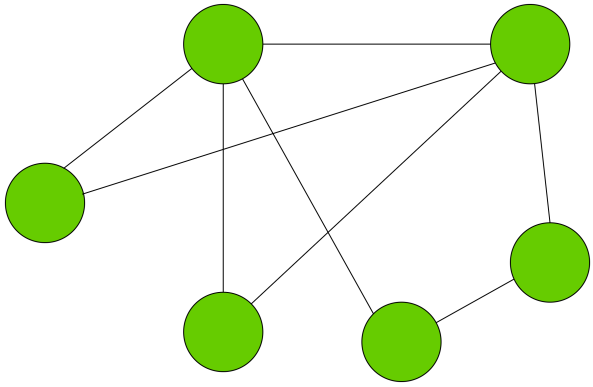


Figure 2.4: Node in graph

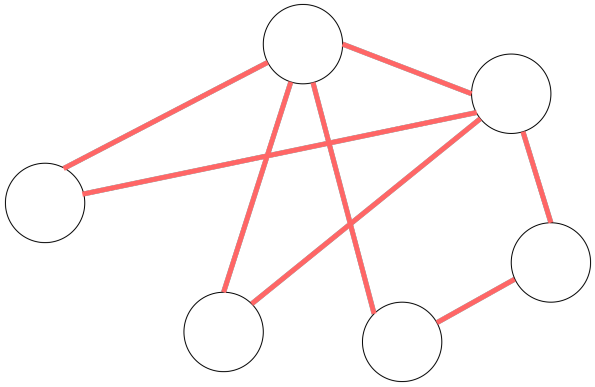


Figure 2.5: Edge in graph

Figure 2.6: Comparison of node attributes in three graphs

In this research scope, we only focus on **undirected graph**.

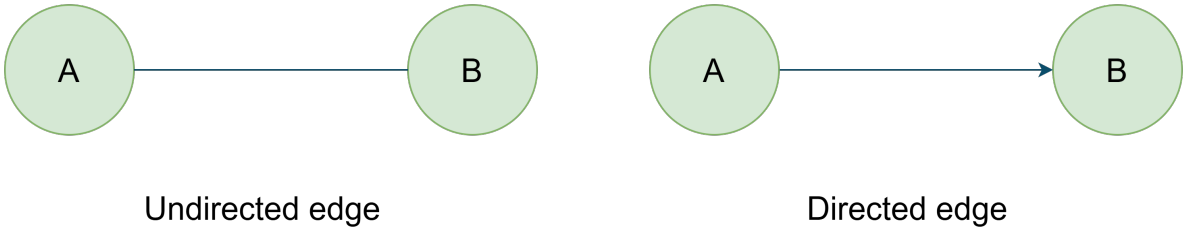


Figure 2.7: Edge representation

2.4.3 Graph representation

- **Adjacency Matrix:** Using two-dimensional (2D) matrix to represent a whole graph, each element in the matrix indicates the presence or absence of an edge between vertices.
- **Adjacency List:** Listing all neighboring vertices for each vertex.
- **Edge List:** The graph is represented as a list of edges, where each edge connects two vertices.

These graph representations are widely used for processing tasks on computers because they optimize data storage and memory usage efficiently. However, for some specific tasks, alternative representations may offer advantages in terms of computational

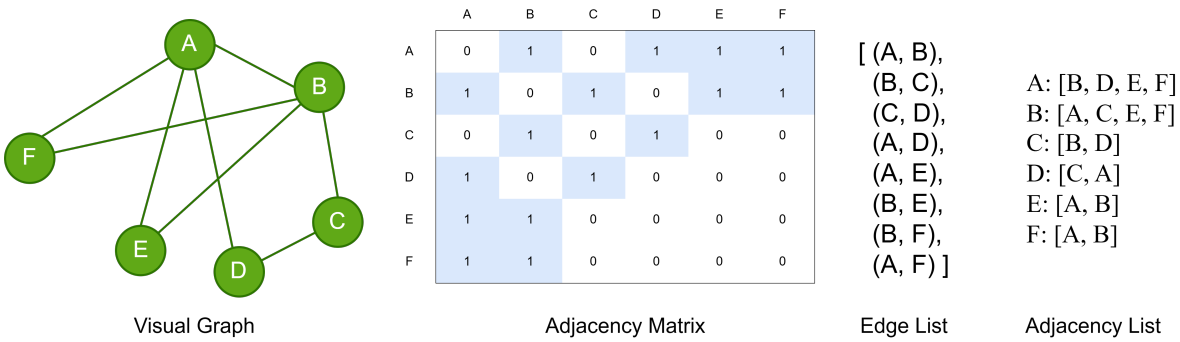


Figure 2.8: Representation of a Graph without Weights

efficiency or suitability for particular algorithms. Hence, it is essential to choose an appropriate format to use for the algorithm, or alternatively, develop a new format tailored for the specific requirements.

2.4.4 Applications of Graph in Machine learning/Deep learning

Graphs are widely used in deep learning due to their significant advantages:

- **Representation of Irregular Data:** In deep learning criteria, Convolutional Neural Networks (CNNs) have been successful in tasks like image classification, semantic segmentation, and machine translation, where data is structured in a grid-like format. These architectures, therefore, leverage local filters with learnable parameters, applying them across all input positions efficiently. However, many real-world problems involve data that cannot be represented in grid structure, such as social networks, biological molecules, brain connectomes. In these scenarios, graphs serve as an ideal representation, , enabling the modeling of complex and irregular relationships that are beyond the scope of traditional grid-based methods.

- **Effiecient Data processing** Graphs can be computed and processed using algorithms specifically designed for graph structures. This allows for the efficient execution of complex computations on large datasets, making the processing both faster and more effective.

- **Visualization** Graphs enable intuitive visualization of data in classification problems, aiding in the interpretation and analysis of relationships. This capability is

particularly useful for advanced deep learning tasks, such as anomaly detection, understanding feature interactions, and uncovering patterns in complex datasets.

With these advantages, graphs are applied in machine learning to process large and complex datasets, enabling prediction and classification. Tasks involving graphs are generally categorized into three main levels: node-level, edge-level, graph-level tasks.

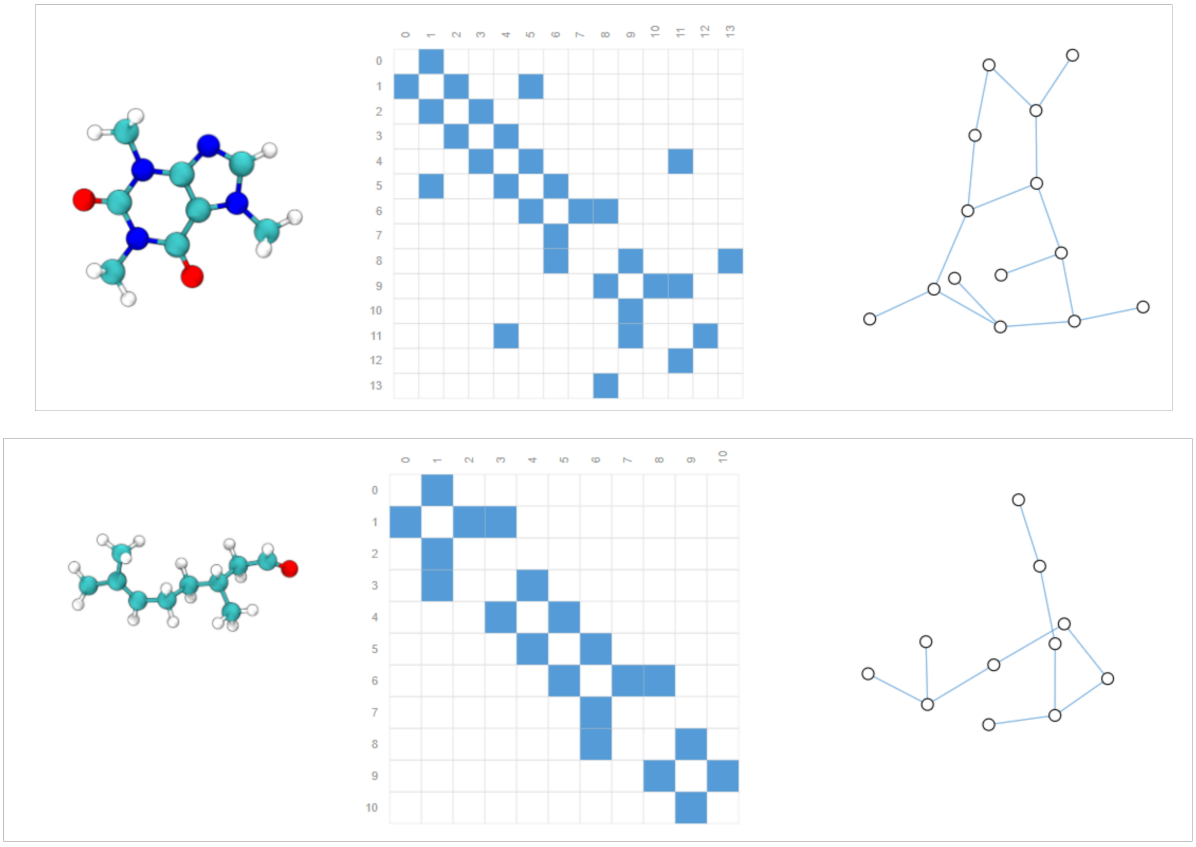


Figure 2.9: Graph can represent structural model such as molecule

2.5 Graph Attention Network (GAT)

2.5.1 GCNN to GAT

Convolutional Neural Networks (CNNs) are effective for tasks with grid-like data structures, such as image classification and segmentation. However, they are not directly applicable to irregular graph-structured data (e.g., social networks, biological networks).

To address this, Graph Neural Networks (GNNs) were developed to handle complex graph structures. Methods like Bruna et al. (2014) and Kipf & Welling (2017) use

spectral representations of graphs, but are limited when applied to graphs with different structures. Non-spectral approaches like MoNet and GraphSAGE define convolutions directly on the graph.

Graph Attention Networks (GAT) were introduced to overcome these limitations. GAT uses a self-attention mechanism to compute node representations by attending to neighboring nodes. This approach enables (1) parallel computation, (2) handling nodes with different degrees, and (3) inductive learning, where the model generalizes to unseen graphs. GAT's flexibility and efficiency make it a strong alternative to traditional GNN methods, particularly in real-world applications with varying graph structures.

2.5.2 A Graph Attention Layer

There are three types of input to calculate in one layer of a Graph Attention Network, given that F is the number of features in one node and F' is those in the next layer.:

1. **Node feature vector** $h \in \mathbb{R}^{1\times F}$, a row vector represents features value in a node, can concatenate all nodes in a graph into a matrix.
2. **Weight matrix** $W \in \mathbb{R}^{F\times F'}$, Weight matrix obtained through the training process.
3. **Attention weight** $a^T \in \mathbb{R}^{1\times 2F'}$, learnable parameter thoughout training process used for self-attention mechanism

Node feature vectors: In models like GAT or GCNN, h exists at each node and represents the unique features of that node in relation to others within the graph. It represents each node's unique attributes, like its properties or role in the graph, and updates by combining information from its neighbors to capture relationships in the graph. In mathematical terms, The element of h can be represented in vector array:

$$\vec{h} = \{h_1, h_2, \ldots, h_N\}$$

Each node has it own vector h, therefore, the whole graph will create a matrix H with **each row represent each node's feature**.

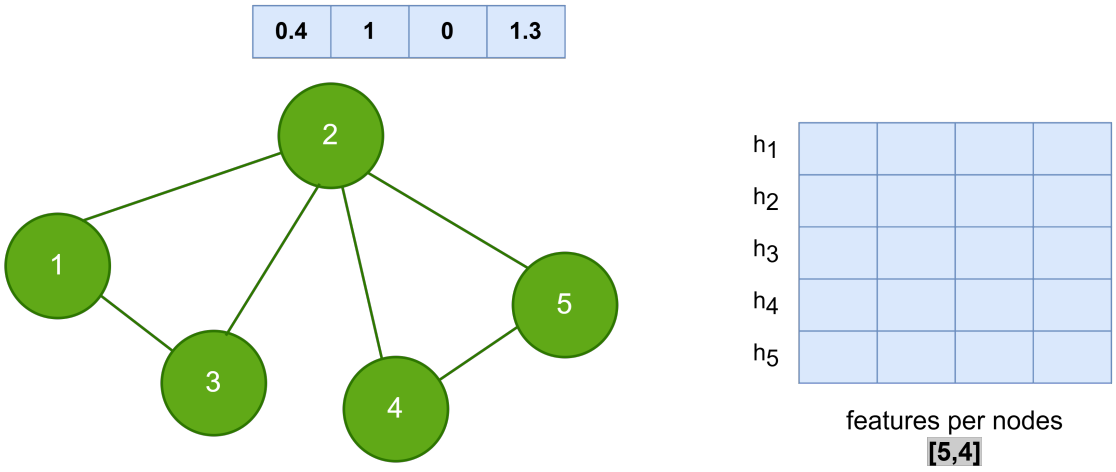


Figure 2.10: Features in Nodes

Weight Matrix W: There is the learnable weight matrix W that transforms the feature vectors of nodes into a new space, enabling the model to extract more meaningful high-level representations. In the GAT model, the weight matrix multiply with a feature vector, called a linear transformation between weights and features node. The result of a linear transformation will assign as z for later calculation.

$$\vec{z_i} = W\vec{h_i} \tag{2.1}$$

Where:

- $\vec{h_i}$: Feature vector of node i
- W: Learnable Weight matrix.
- $\vec{z_i}$: result of the linear transformation

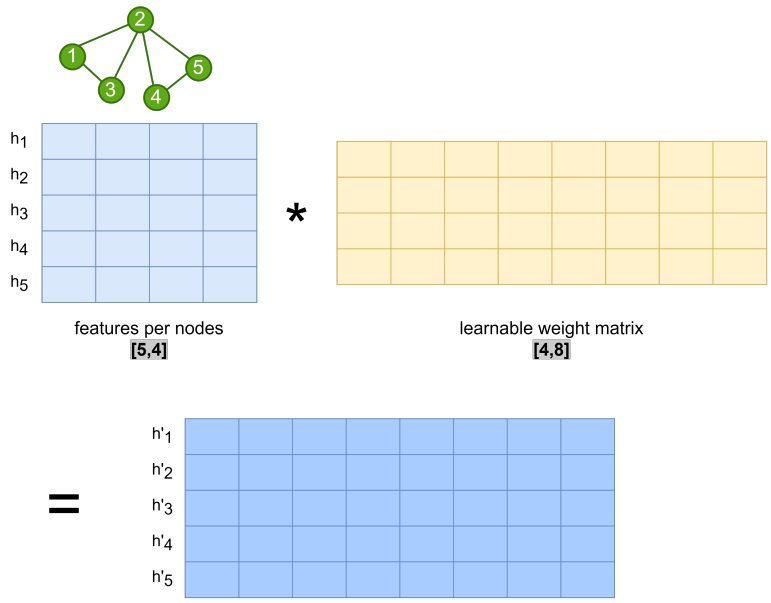


Figure 2.11: Matrix Multiplication represent Graph features

Self-attention mechanism The basic idea of attention in graph neural networks, particularly in GAT, is to learn the importance of neighboring node features for updating a target node's representation. This importance is quantified as the attention coefficient, which is computed for each neighboring node and determines the weight assigned to its features during aggregation. This mechanism allows the model to focus on the most relevant neighbors dynamically, enhancing its ability to capture meaningful relationships in the graph. Formula to calculate attention coefficient:

$$e_{ij} = a(W\vec{h_i}, W\vec{h_j}) \tag{2.2}$$

Where:

- e_{ij} is the attention coefficient of node j's features to node i.
- $W\vec{h_i}$, $W\vec{h_j}$ is the result of the (2.1).
- a is the attention weight.

Softmax To make coefficients easily comparable across different nodes, we normalize them across all choices of j using the softmax function, before exponential the attention coefficient values, all coefficient must filted through a non-activation function, such as LeakyRELU (with slope=0.2):

$$\alpha_{ij} = \text{softmax}(e_{ij}) = \frac{\exp(\text{LeakyRELU}(e_{ij}))}{\sum_{k \in \mathcal{N}_i} \exp(\text{LeakyRELU}(e_{ik}))}. \tag{2.3}$$

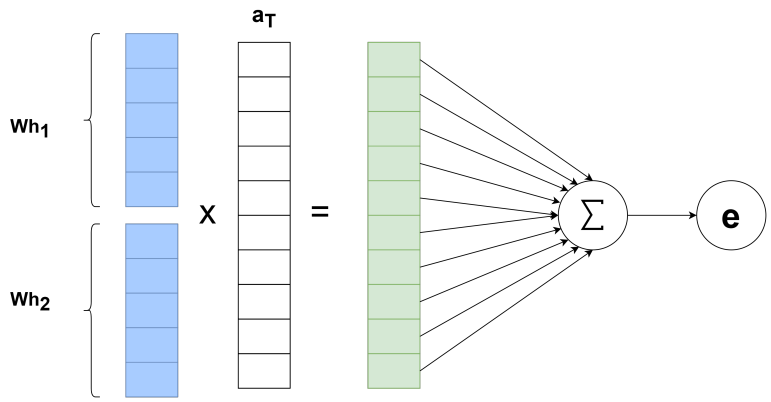


Figure 2.12: Calculation of Attention coefficient

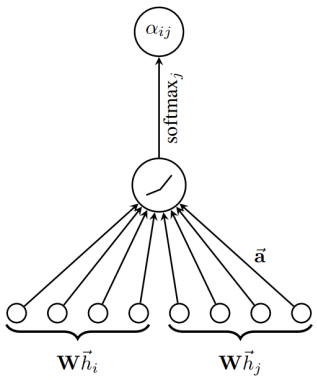


Figure 2.13: Softmax flow with LeakyRELU

Aggregator Using the result of softmax, we combine with the Wh to get the feature matrix for the next layer. In this research scope, we do not implement the multi-head attention hence the default value of head = 1, the equation is:

$$\vec{\mathbf{h}}_i' = \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij} W\vec{\mathbf{h}}_j\right) \tag{2.4}$$

Where σ is a **non-linear activation function** (e.g RELU, LeakyRELU, sigmoid,...)

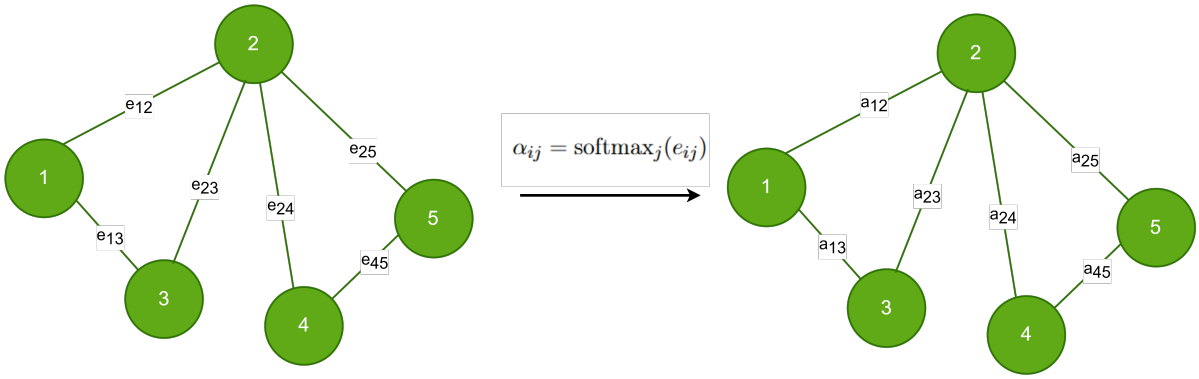


Figure 2.14: Attention Coefficient and Softmax

2.5.3 Graph Attention Network Applications

Inductive Tasks

- **Abnomaly Detection**: GAT detects anomalies in graphs, such as identifying unusual activity in network traffic or detecting fraud in financial transactions. (Zhao

et al., 2020 [10])

- **Recommendation Systems:** GAT models user preferences to improve recommendation accuracy by identifying relevant products or services or neural social recommendation system. (Mu et al., 2019 [4])
- **Text Classification:** GAT represents text as graphs and applies attention to key words, enhancing the accuracy of text classification tasks (Li et al., 2024 [3])

Transductive Tasks

- **Node Classification:** GAT effectively classifies nodes in graphs, such as citation networks and social networks, by leveraging attention to prioritize influential neighbors. (Veličković et al., 2018 [7]
- **Image Classification:** GAT is used for image segmentation by modeling images as graphs and focusing on significant regions through attention. (Lei et al., 2022 [2])

Besides its application, it is believed that the performance of GAT is currently better than GCN. Veličković reported the classification accuracy results as follows, on Cora, Citeseer, and Pubmed datasets. GCN-64 corresponds to the best GCN result computing 64 hidden features (using ReLU or ELU):

Model	**Cora (%)**	**Citeseer (%)**	**Pubmed (%)**
GCN-64	81.4 ± 0.5	70.9 ± 0.5	79.0 ± 0.3
GAT (ours)	83.0 ± 0.7	72.5 ± 0.7	79.0 ± 0.3

Table 2.2: Comparison of classification accuracies for GCN-64 and GAT (ours) on 3 dataset

2.6 Related Work

Existing efforts to accelerate GNN and GAT inference in FPGA often begin with algorithmic optimizations before developing corresponding hardware architectures.

LW-GCN [6] is an FPGA-based accelerator for GCNs designed for high energy efficiency and low latency. It preprocesses sparse matrices into a ”Packet-level Column-only Coordinate List” (PCOO) format to reduce storage and bandwidth requirements while balancing workloads across processing elements (PEs). Its hardware features a multi-bank dense memory with data replication to avoid collisions and a round-robin method to distribute non-zero elements evenly among PEs. Rows are concatenated before assignment to PEs to handle variations in non-zero elements efficiently.

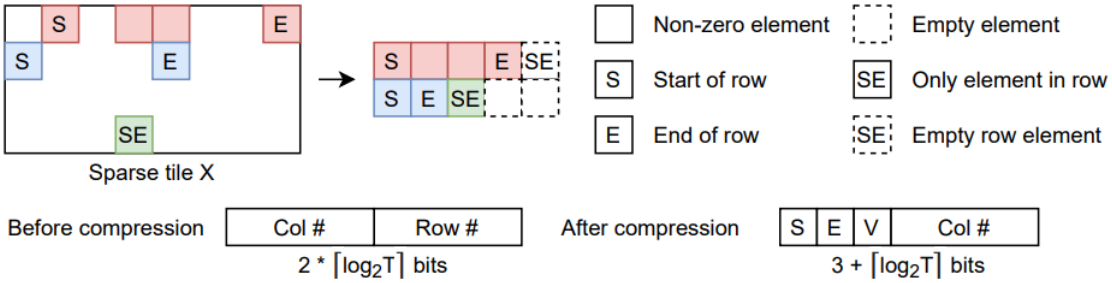


Figure 2.15: LW-GCN Packet Column-only coordinate list format

S-GAT [9] is an FPGA-based accelerator for GATs that uses model compression, feature quantization, and shift addition units (SAUs) to reduce computation and memory requirements. It implements a universal hardware pipeline and avoids reliance on DSPs. Tested on the Inspur F10A (Intel Arria 10 GX1150), S-GAT achieves up to 7.34x speedup over an Nvidia Tesla V100, 593x over a Xeon CPU Gold 5115, and improves energy efficiency by 48x and 2400x, respectively. However, S-GAT faces challenges in ignoring preprocessing and CPU communication overhead, and its PE units do not fully utilize graph sparsity for efficient matrix multiplication.

FTW-GAT [1] addresses the challenges of GAT inference by using ternary weight networks (TWNs), which reduce memory usage, simplify processing elements, and eliminate the need for DSPs. It also uses multi-level pipelines and specialized computing units to improve performance. However, due to the complexity of GAT inference and its data dependencies, pipeline stalls still occur frequently during execution.

SH-GAT [8] addresses the limitations of previous GAT accelerators by introducing hardware-friendly optimizations, including split weights and softmax approximation to reduce computational complexity, and a load-balanced SPMM kernel to maximize par-

allelism and data throughput. To mitigate pipeline stalls caused by irregular memory access, it employs preprocessing techniques that pre-fetch source and neighbor nodes. Evaluated on the Xilinx Alveo U280, SH-GAT achieves up to 3283×, 13×, and 2.3× speedups compared to CPUs, GPUs, and prior FPGA-based accelerators, respectively. However, the research does not point out the quantization method to retain the accuracy of softmax. To address these issues, our design focuses on enhancing computational parallelism and accelerating inference while minimizing the excessive use of storage resources.

Accelerator	**Key Features**	**FPGA Board**	**Performance**
LW-GCN	PCOO format, workload balancing, round-robin allocation, multi-bank memory	Xilinx Kintex-7 325T	Up to 60x faster than CPU, 12x GPU, 1.7x AWB-GCN; Power efficiency: 912x CPU, 511x GPU, 3.87x AWB-GCN
S-GAT	Model compression, feature quantization, SAUs for shift-based multiplication	Inspur F10A (Intel Arria 10 GX1150)	7.34x faster than Nvidia Tesla V100
SH-GAT	Split weights, softmax approximation, load-balanced SPMM, pre-fetching to address memory irregularities	Xilinx Alveo U280	Up to 3283x faster than CPU, 13x GPU, 2.3x SOTA FPGA
FTW-GAT	Ternary weight quantization, operation fusion, multi-level pipelining, graph partitioning	Xilinx VCU128	Performance: 390x CPU, 17x GPU, 1.4x prior GAT accelerator; Energy efficiency: 4007x CPU, 261x GPU, 3.1x prior accelerator

Table 2.3: Comparison of Related Work on FPGA-based GCN Accelerators

Chapter 3

Related Architecture

This chapter details the optimization of the GAT algorithm for FPGA implementation. It covers the GAT algorithm optimization, focusing on reducing computational complexity and improving parallelism. We introduce a new data format, GCSR (Graph Compressed Sparse Row), for efficient graph representation. Data preprocessing steps are discussed to prepare input for hardware processing. Finally, we apply quantization techniques to reduce model size and computation demands, enabling efficient GAT execution on FPGA.

$$e_{ij}^{(l)} = \text{LeakyReLU}\left(a^{(l)T}\frac{z_i^{(l)}}{\|z_i^{(l)}\|} \cdot \frac{z_j^{(l)}}{\|z_j^{(l)}\|}\right)$$

3.1 GAT Algorithm Optimization

Graph Attention Network (GATs) are powerful models for graph-based deep learning tasks. However, in practice, large-scale graph datasets may consume significant amount of resources in hardware, especially on platforms like FPGAs. Hence, it is essential to improve its graph data format as well as modify the algorithm's representation to make GAT more "hardware-friendly".

The computational resources in hardware are often insufficient to handle a large volume of calculations or operations and different bit formats efficiently. In the inference models, model fine-tuning is allowed, though the trained model must ensure accuracy on the dataset. In this research, we proposed changing the computation method to reduce

the computational load, as well as reuse modules and minimize the modification of bit size of a data unit in memory.

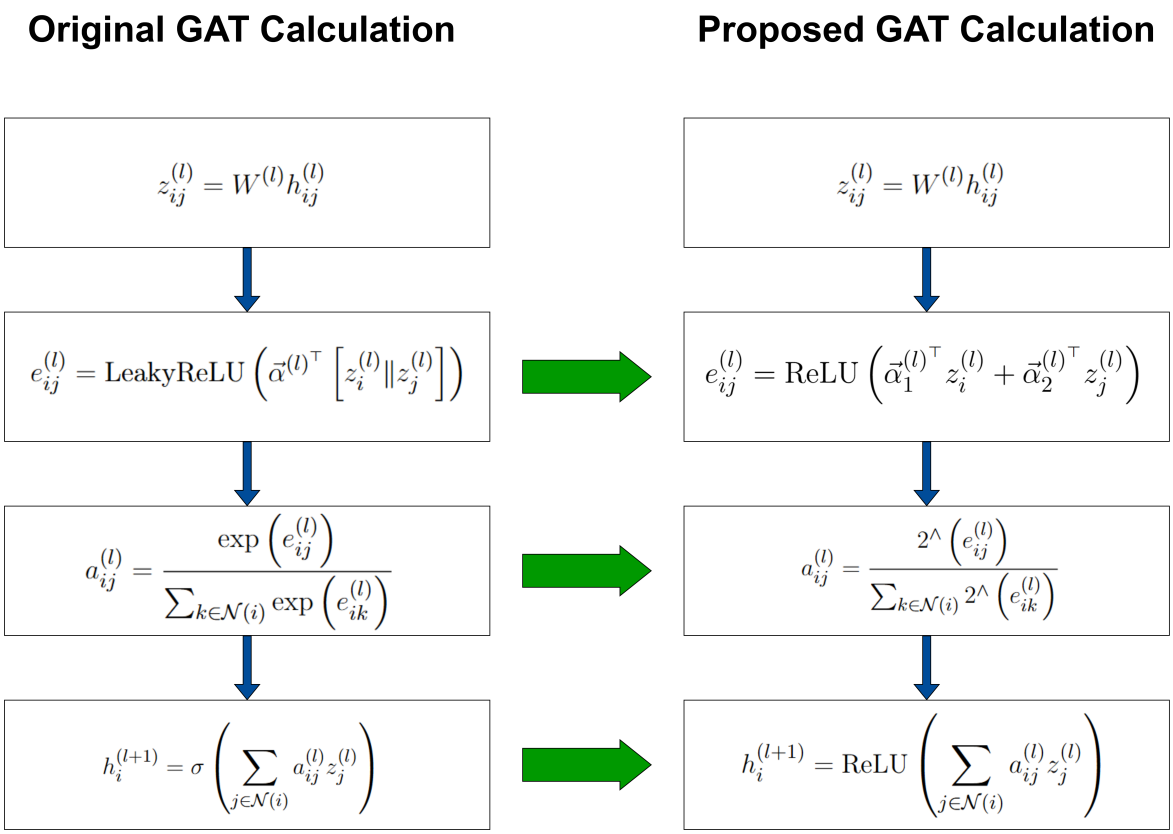


Figure 3.1: Proposed Modification in GAT Algorithm

Exponentiation Approximation

In the softmax calculation, the exponential function uses the constant e (approximately 2.718) and exponentiation by x (8 bits), making the computation resource-intensive when using floating-point precision. Additionally, implementing the exponential function with constant e is relatively complex in hardware. Therefore, we currently use **exponentiation with base 2 (2^x)** to reduce the complexity of the module, taking advantage of the simple and time-saving shift-bit operations that are easier to implement in hardware, while maintaining a reasonable level of approximation correctness.

Activation functions

In the attention calculations (steps 2 and 4), the original GAT formula uses LeakyReLU as the default activation function to retain necessary negative values. However, we will change all of them to ReLU to allow module reuse across calculation steps.

More specifically, changing to ReLU is necessary for softmax calculation. Currently, our proposed architecture uses signed 8-bit data, which ranges from -127 to 127. Since softmax involves exponentiation, the result of the bit-shifting operation (2^x) on 8-bit data requires a maximum of 128 bits. If LeakyReLU is used in the previous step, negative values in the data are retained, meaning that the result 2^x will be formatted in both fixed-point Q128.0 and Q0.128. As a result, the fixed-point format of $softmax(e)$ will be Q128.128, extending up to 256 bits. Therefore, Using ReLU is expected to eliminate negative values, thus keeping the fixed-point format as Q128.0 throughout the calculation steps.

Attention Coefficient

In the initial computation of attention coefficients (step 2), for a single subgraph, z_i and z_j are concatenated together and then multiplied by the attention weights. In the subgraph, z_i is considered the source node, which is connected to many other neighboring nodes z_j. Therefore, the vector multiplication of the concatenated vectors will repeatedly multiply the source node with the first indices of the attention weights. This repetition of the computation consumes significant computational resources. As a result, our study

proposes splitting the attention weight a into two weights, a_1 and a_2 , to perform separate multiplications with each z. The multiplication for the source node can then be stored and reused in the adder unit after the multiplication unit has completed its operation.

Figure 3.2 is an example of the calculation of concatenation a and a proposed method to divide it into a_1 and a_2, the example provides a subgraph where node 1 is the source node, and nodes 2, 3, 4, and 5 are the neighboring nodes. It is obvious that $z_1 \times a_1$ will repeatedly multiply across the nodes when using the default formula. The proposed solution is to separate it into independent a_i weights and multiply them independently.

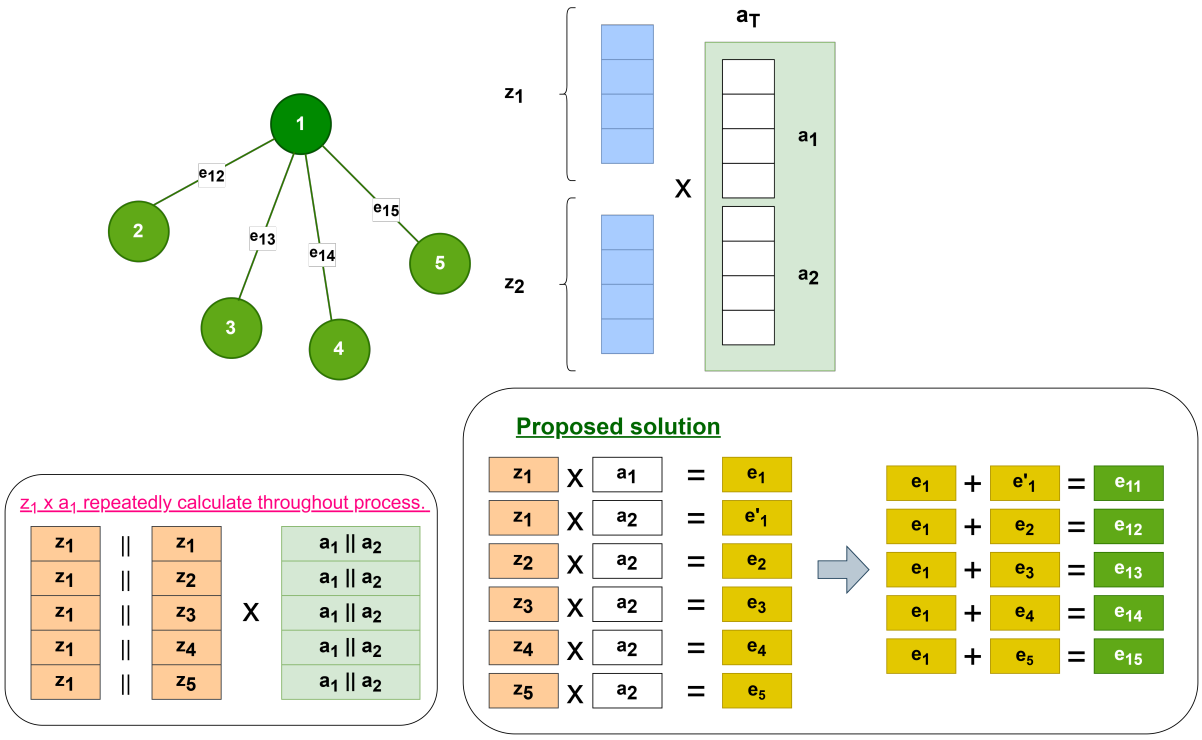


Figure 3.2: Issue when concatenating nodes and Proposed solution

3.2 Data Preprocessing

When performing calculations (parallel or pipelining) in hardware, results such as z_{ij} are saved and fetched from memory. If these values are unstructured, which means values are stored sparsely and lack connectivity according to the edges, multiple control signals are required throughout modules to manage and coordinate the linked edges effectively, as a result, cost a considerable amount of hardware resources. Additionally, if the z is sparse and each z_{ij} retrieved in an unordered way, other calculation modules need to wait for z_{ij} to be ready, so it may cause some timing delay for the system, called **pipeline stall**.

Since the attention mechanism inherently relies on the relationships between nodes, it is necessary to track the adjacent nodes, or acknowledge the connection of relevant nodes before transferring data to hardware implementation.

The software implementation of data preprocessing is necessary because of those aforementioned issues in hardware. The purpose of this work is to arrange the components of the source node and its neighboring nodes together, forming a subgraph (its head is the source node). This approach simplifies the management of data transactions

as they pass through computational modules, as illustrated in Figure 3.3. In this figure, (a) represents the non-preprocessed flow, which faces pipeline stalls, whereas (b) shows the preprocessed flow, where data is grouped to avoid such issues.

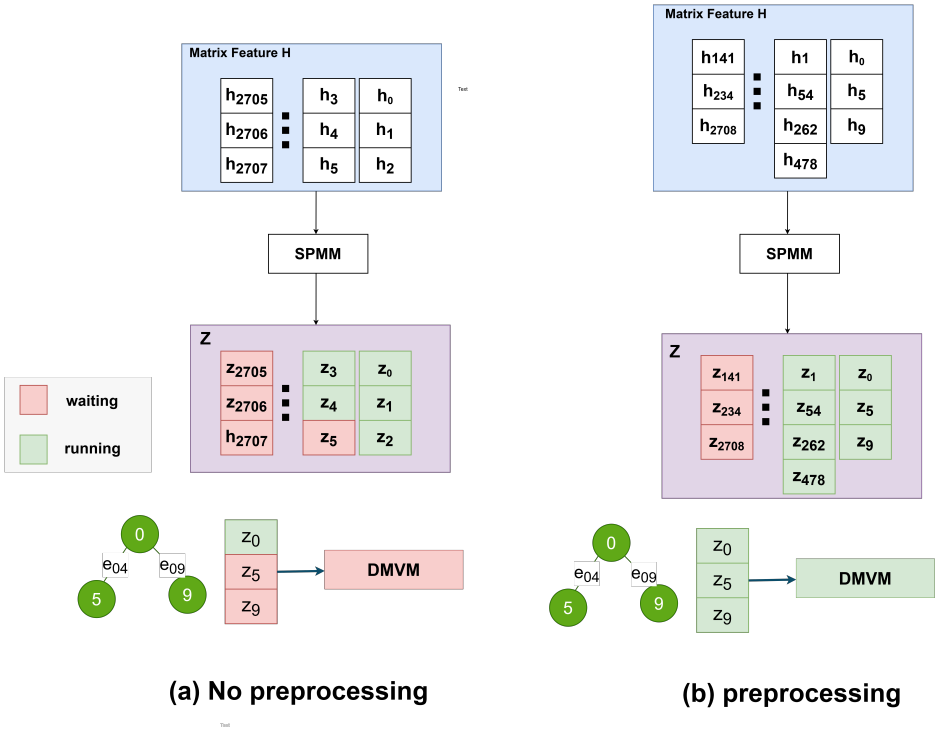


Figure 3.3: Illustration of no preprocessing and preprocessing data flow

As proposed, the preprocessing will focus on grouping the graphs into subgraphs. Therefore, the input graph data, represented by the feature matrix H, must be reorganized. The vectors $\vec{h}$ are grouped together, with the source node leading each group. Hence, this processed subgraph format will be the data transferred to the Programmable Logic part in the hardware.

More specifically, preprocessing the data into subgraphs is illustrated in Figure 3.4, which shows the extraction of a 4-node graph into 4 subgraphs. Each subgraph begins with the feature of the source node at the head, followed by the features of its neighboring nodes in the subsequent rows.

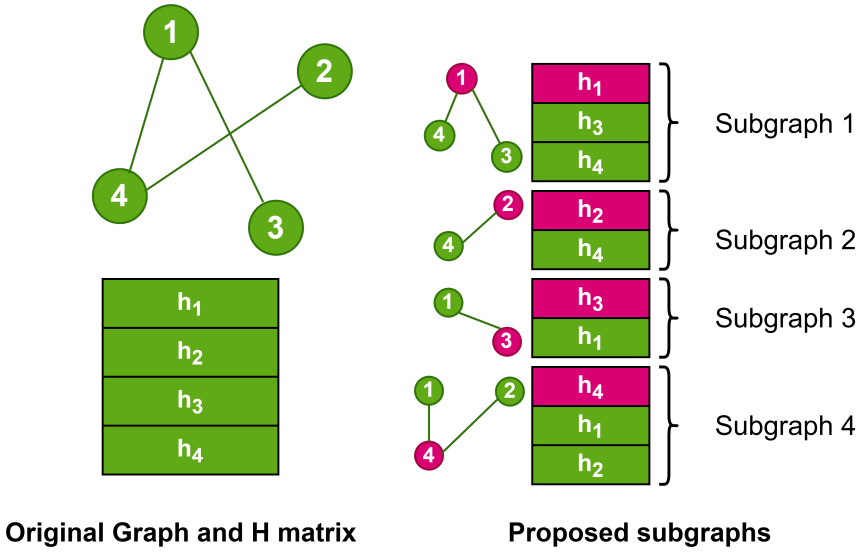


Figure 3.4: Demonstration of the extraction of a simple graph into subgraphs

3.3 Graph Compressed Sparse Row - GCSR

Graph feature matrices and adjacency matrices are often sparse (> 50% entries being zero). In hardware implementation, directly storing and processing sparse matrices would cost a lot of memory space and computational resources.

In actual datasets of graph inferences, it is clarified that most of the node feature matrices are sparsed:

Dataset	Nodes	Edges	Input Feature	Classes	Feature Density	Edge Density	Weight Density
Cora	2708	10,556	1433	7	1.3%	0.14%	100%
CiteSeer	3327	9104	3703	6	0.8%	0.08%	100%
PubMed	19,717	88,648	500	3	10.4%	0.02%	100%

Table 3.1: Dataset Information with Nodes, Edges, and Features

In this research, we proposed a new format for data of graph, by embedding the value and index into different types of arrays. The format called Graph Compressed Spare Row (GCSR), which will focus only on non-zero elements along with their positions in the graph. It is expected to reduce a significant amount of memory usage and computational overhead.

The GCSR format involves 3 main components:

- **col_index**: Store the column indices of non-zero elements in the feature matrix.

- **value**: contain the values of the non-zero elements.
- **node_info**: this can be the metadata for extra-information of the nodes. There are 3 information compressed into binary format:
  - row_length: number of the non-zero feature values in one node (row is a h in the whole feature matrix)
  - node_flag: identifies whether a node is a source node (=1) or a neighbor node (=0).
  - num_of_nodes: the quantity of nodes (or rows) in a subgraph.

To be more specific, node_info can be represented in binary format, for example: Given a node feature matrix H of a **3-node** subgraph :

$$\mathbf{H} = \begin{bmatrix} 0 & 2 & 1 & 3 & 0 & 9 \\ 4 & 0 & 3 & 0 & 0 & 5 \\ 8 & 6 & 0 & 3 & 0 & 4 \end{bmatrix}$$

In first row $\vec{h_0} = [0, 2, 1, 3, 0, 9]$, node_info will be:

- row_length: **4**, for there are 4 non-zero elements in the vector.
- num_of_nodes: **3**, for the sub-graph comprises of 3 node feature vectors.
- node_flag: **1**, for first row of the sub-graph always be the source node.

By concatenating the three values mentioned above, we obtain one element of the node_info for the first node:

4‖**3**‖**1**

equivalent value in binary format:

100‖**011**‖**1**

In the second row, $\vec{h_1} = [4, 0, 3, 0, 0, 5]$:

- row_length: **3**
- num_of_nodes: **3**

- node_flag: **0**.

Hence, node_info will be:

$$\mathbf{3||3||0 \rightarrow 011||011||0}$$

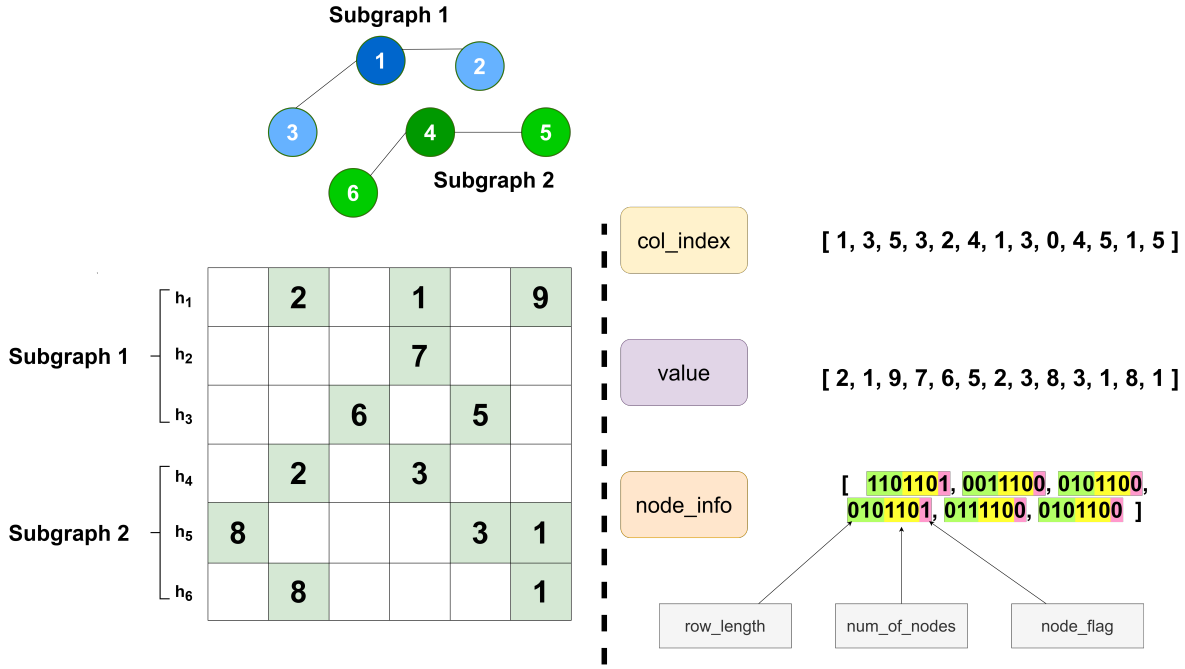


Figure 3.5: Proposed Modification in GAT Algorithm

3.4 PCOO - GCSR Comparison

3.5 Quantization

Floating Point in GAT and Related issues

Graph Attention Networks (GAT), rely on high-dimensional matrix operations (multiplications, softmax,…) and the computation of attention coefficients across edges. These operations often use FP32 due to its precision, which minimizes numerical errors during backpropagation, matrix multiplications and ensure accurate attention distribution across edges. However, this high level of precision comes with trade-offs in terms of storage especially when scaling to larger graphs or deploying the model in resource-constrained environments. In addition to **storage** resource concerns, computations using FP32 (32-

bit floating-point format) also require significant **resources** and **time**.

Figure 3.6: Components of 32-bit floating-point format

Optimization Possibility

While FP32 provides high precision, this level of granularity is often excessive for the practical ranges of weights or activations encountered during training and inference. **Weights** in many deep learning models, including GAT, are normalized during training and converge within relatively **narrow ranges** (e.g., -1 to 1). As a result, the extra precision offered by FP32 becomes redundant.

Unlike floating-point numbers, **fixed-point numbers** are quite similar to integers (since all values share a common decimal point position, which is implicitly understood and omitted). The advantage of fixed-point representation is that it eliminates the need to compute two separate components of floating-point numbers (the exponent and the mantissa). This reduction leads to **lower resource usage** in terms of energy and space for operations like multiplication or addition. This process is easily implemented through a technique called **quantization**.

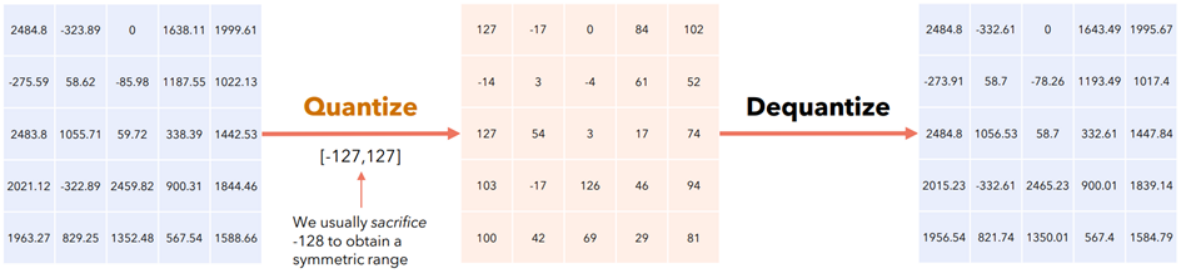


Figure 3.7: Quantization and Dequantization Process

Quantization Techniques in Machine Learning

Quantization is a widely used technique in deep learning to reduce the size of models, lower inference latency, and speed up computations, especially in resource-constrained

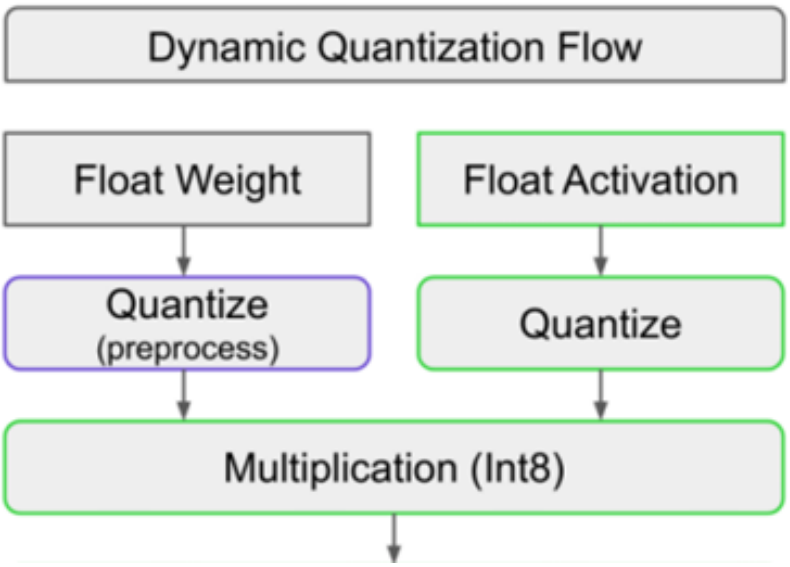


Figure 3.8: Dynamic Quantization Process

environments like edge devices, or hardware accelerators such as **FPGAs**. In common, reducing from FP32 to INT8 achieves a **4x reduction in storage** requirements and delivers **1.5x to 4x latency improvements** on CPUs and specialized accelerators.

- **Dynamic Quantization**

  In the **inference** of neural network using dynamic quantization, it uses **integer ops** as many as possible. The **weights** were quantized into integers prior to the inference runtime. However, since the neural networks does not know the scales and the zero point for the output or activation tensors, it has to be a floating point tensor. Then once we could find the (α,β) for the tensor, compute the **"scaling factor": scale and the zero point**, and quantize the floating point tensor to the nearest integer and storing it tensor dynamically during **runtime**. This is why it is called **dynamic quantization**.

  However, in most of the cases, the **overhead of dynamic quantization**, compared to **static quantization**, is that the scales and zero points for all the activation tensors and the quantized activation tensors have to be computed dynamically from the floating point activation tensors.

- **Post Training Static Quantization**

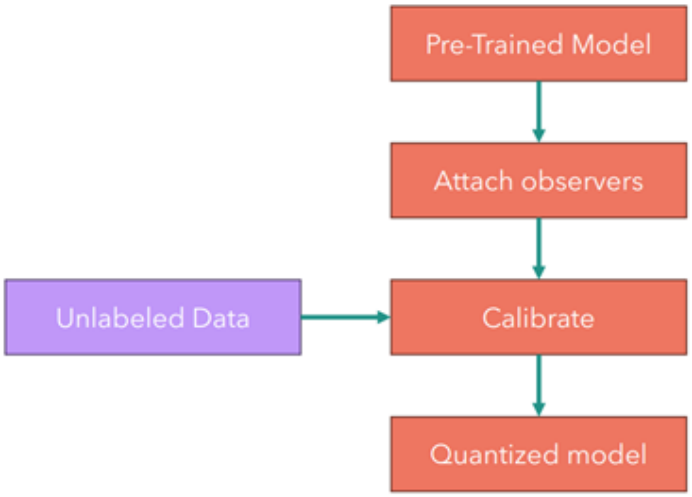


Figure 3.9: Static Quantization Process

What's different to dynamic quantization is that, **static quantization** determines the scales and zero points for all the activation tensors are **pre-computed**. Therefore, the overhead of computing the scales and zero points is eliminated. The activation tensors could be stored as integer tensors in the memory without having to be converted from floating point tensors.

The way to determine the scales and zero points for all the activation tensors is simple. Given a floating point neural network, we would just have to run the neural network using some representative unlabeled data **(calibration dataset)**, collect the distribution statistics for all the activation layers. Then we could use the distribution statistics to compute the scales and zero points using the mathematical equations.

During **inference**, because all the computations were conducted seamlessly using integer ops. The only short-coming is while static quantization improves model size and speed, it can lead to a **loss in accuracy** compared to **Quantize Aware Training (QAT),** as the model hasn't been adjusted for the quantization error, and the inference accuracy will be harmed.

- **Quantization Aware Training**

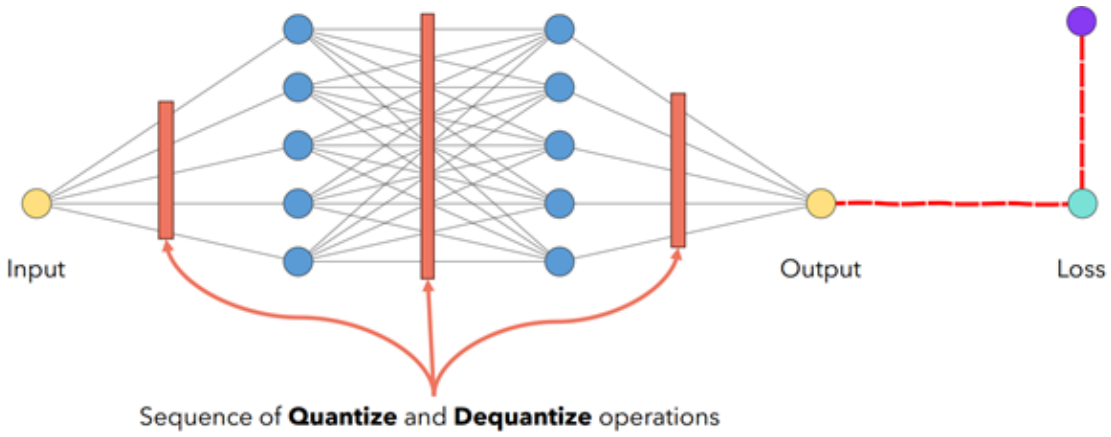


Figure 3.10: Quantization Aware Training Process

Quantization used in neural networks introduces information loss and therefore the inference accuracies from the quantized integer models are inevitably lower than that from the floating point models. Such information loss is due to that the floating points after quantization and de-quantization is not exactly recoverable. Mathematically, it means

$$x = f_d\big(f_q(x, s_x, z_x), s_x, z_x\big) + \Delta_x$$

where

- f_q is the quantization function
- f_d is the de-quantization function
- Δx is an unknown small value
- s is a scale point
- z is a zero point

If $\Delta x=0$, the inference accuracies from the quantized integer models would be exactly the same as the inference accuracies from the floating point models. Unfortunately, it is not. To **address this issue**, the idea of **quantization aware training** is to ask the neural networks to take the effect of such information loss into account during training. More specifically, during neural networks training, all the activation or output tensors and weight tensors are variables, so in quantization

aware training, we added a quantization and a de-quantization layer for each of the variable tensors. Mathematically, it means

$$\hat{x} = f_d\big(f_q(x, s_x, z_x), s_x, z_x\big) = s_x\Bigg(\text{clip}\Big(\text{round}\big(\frac{1}{s_x}x + z_x\big), \alpha_q, \beta_q\Big) - z_x\Bigg)$$

where the scale and zero point could be collected using the same method we discussed in static quantization. For example, a floating point 242.4 after quantization and de-quantization would become 237.5, but they are still very close. All the data types for the quantized tensors are still floating point tensors and the training is supposed to be done normally as if the quantization and de-quantization layers were not existed

Applied Techniques and Explanation

- **Quantization**
  - **Quantization Aware Training (QAT)**
  - **Explanation**

    Quantization Aware Training (QAT) is a technique where the quantization process is integrated into the model's training procedure. During QAT, compared to 2 other quantizations, all parameters are simulated to be quantized (i.e., reduced to lower precision values such as integers), while still maintaining the full precision of floating-point numbers during backpropagation. This allows the model to adjust and adapt to the quantization error during training, enabling it to minimize accuracy loss when quantization is eventually applied for inference.

- **Calibration**
  - **Max-Min Calibration**
  - **Explanation**

    Max-Min Calibration is a method used in to determine the optimal scaling factors. It works by observing the maximum and minimum values and the

scaling factors are then calculated by finding the range (difference between the maximum and minimum values) of these. This technique is simple and effective, as it captures the extreme values of the tensors, ensuring that the entire range of values is represented within the quantized range.

- **Range Mapping**

Figure 3.11: Range Mapping Symmetric and Assymetric

  - **Symmetric**
  - **Explanation**

    Symmetric range mapping is a technique where the range of the values for the quantized tensor is mapped symmetrically around zero. This means that both the positive and negative values are treated equally when scaling the range. For example, if the data has a range from -5 to 5, symmetric mapping would treat this as a range from -5 to 5, without any offset or shift. Symmetric mapping reduces the risk of losing significant data range when quantizing, especially when the data distribution is balanced across both positive and negative values.

Chapter 4

Overall Architecture

Based on the ideas and works discussed earlier, this section explains the design of the proposed architecture for speeding up the GAT algorithm on the FPGA platform. This section describes the system architecture with a top-down approach. Moreover, the general design and principle of components are also illustrated here.

From the analysis of the **Cora** dataset, the Feature Matrix H is significant sparsity with a density of only 1.3%. To efficiently manage this, our proposed architecture will represent Feature Matrix H as the **GCSR** format. Additionally, through our proposed pre-processing step, the Feature Matrix H will be reorganized into 2,708 consecutive subgraphs, corresponding to the 2,708 nodes in the **Cora** dataset. Each subgraph captures the relationships between a source node and its neighboring nodes, resulting in a total matrix size of 13,264 × 1,433.

4.1 General Architecture

The general architecture consists of two main parts: **PS** (Processing System) and **PL** (Programmable Logic). This architecture exploits the parallel computation strength of **PL** to process the GAT model. Figure 4.1 shows an overall design.

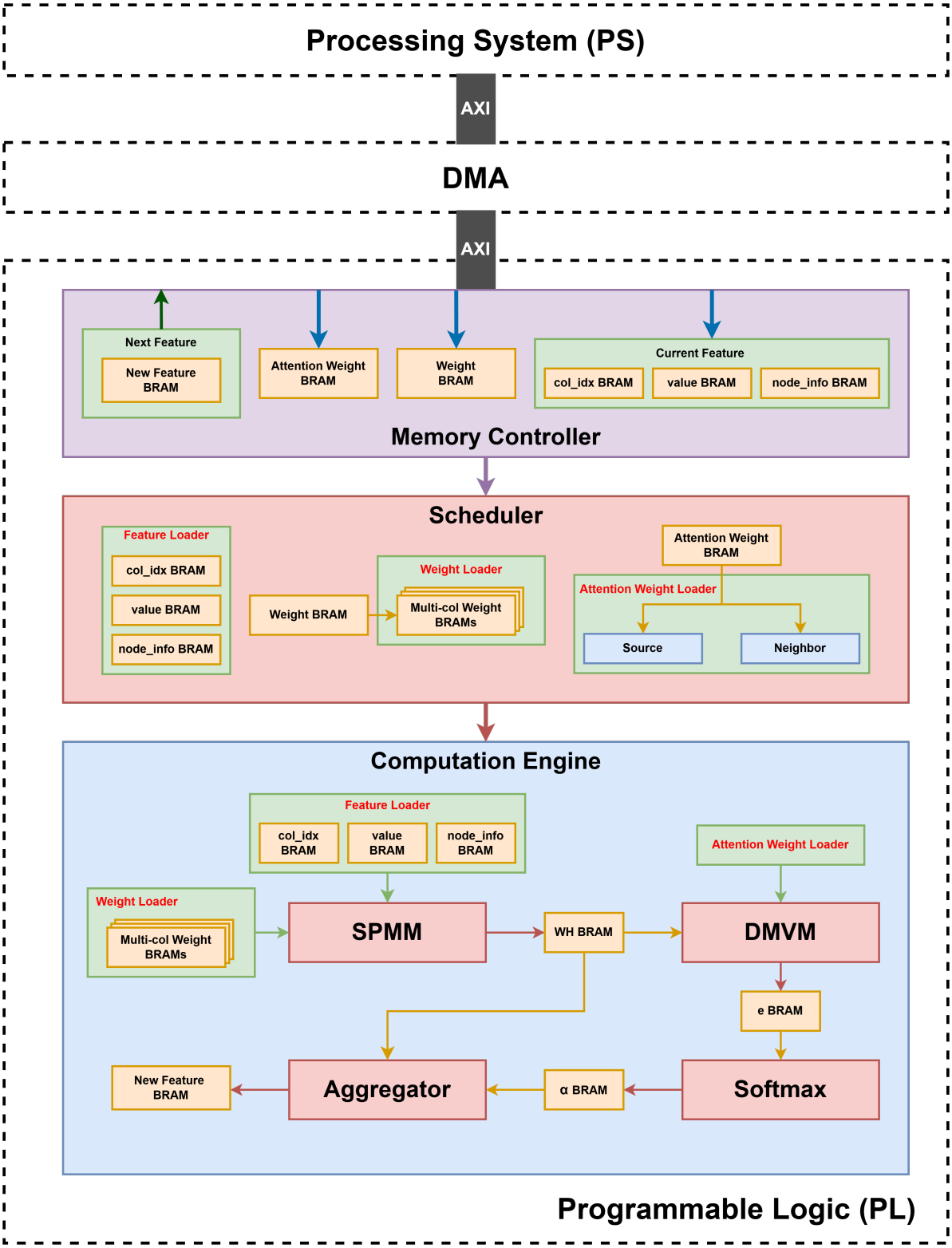


Figure 4.1: General Architecture of GAT accelerator

Our architecture consists of several key components:

1. **PS** (Processing System): The **PS** is responsible for executing software applications, including training the GAT algorithm, preprocessing the Feature Matrix H into

the **GCSR** format, reorganizing it into multiple subgraphs, and transferring the processed data to the **PL** via **CDMA** (Central Direct Memory Access) using **AXI** (Advanced eXtensible Interface) protocol.

2. **Memory Controller:** The **Memory Controller** acts as the storage interface between the **PS** and **PL**. After the **PS** prepares the three matrices: Feature Matrix H, Weight Matrix W, and Attention Weight a, they are transferred to the **PL** and stored in their respective **BRAMs**. Once the **PL** computes the Feature Matrix H for the next layer, it is stored in the **BRAM**, enabling the **PS** to read, quantize, and preprocess it for subsequent layer calculations.

    - Current Feature Matrix H: Stored across **col_idx BRAM**, **value BRAM**, and **node_info BRAM**, following the **GCSR** format.
    - Weight Matrix W: Stored in **Weight BRAM**.
    - Attention Weight a: Stored in **a BRAM**.
    - Next Feature Matrix H: Stored in **Next Feature BRAM**.

3. **Scheduler:** Once the data is transferred from the **PS** and stored in **BRAMs** in **Memory Controller**, the **Scheduler** handles preprocessing and data preparation for subsequent computations. Based on the number of hidden channels in the next layer (16 hidden channels in this work), the *Weight Matrix* W retrieved from **Weight BRAM** is divided into 16 separate **BRAMs**, each representing a column of the *Weight Matrix*. For the Attention Weight a, the **Scheduler** generates into two sub-matrices: *source* and *neighbor* to facilitate further computations.

4. **SPMM** (Sparse Matrix Multiplication): From *Feature Loader* and *Weight Loader* in **Scheduler**, **SPMM** performs the matrix-vector multiplication of W and H, storing the resulting vector $W\vec{h}$ in **WH BRAM**.

5. **DMVM** (Dense Matrix Vector Multiplication): This module implements the **Self-Attention** equation from 2.2. It retrieves the $W\vec{h}$ vector from **WH BRAM** and multiplies it with the corresponding *Attention Weight*, which is determined based on whether the $W\vec{h}$ vector corresponds to a *source* node or a *neighbor* node. Upon

completing each calculation, all attention coefficients for each subgraph are concatenated and stored in **e BRAM**.

6. **Softmax:** This module normalizes all attention coefficients within a subgraph which are retrieved from the **e BRAM** to ensure they are comparable across different nodes. The resulting normalized attention coefficients α are then stored in the α **BRAM**.

7. **Aggregator:** The **Aggregator** retrieves the normalized attention coefficients from α **BRAM** and combines them with the corresponding feature vectors from **WH BRAM** to compute the updated node features for the next layer. The aggregated feature vectors are then stored in the **New Feature BRAM**, completing the feature update process for the current layer.

4.2 Scheduler & SPMM

SPMM (Sparse Matrix-Vector Multiplication) is responsible for computing the multiplication between the *Feature Matrix* H and the *Weight Matrix* W, with data retrieved from the loaders managed by the **Scheduler**. The following equation illustrates how we computed it in hardware:

$$z \mathrel{+}= value \times Weight[col\_idx] \tag{4.1}$$

It consists of the following components:

- **Feature Loader:** This module retrieves 3 information from *Feature Matrix* based on **GSCR** format: *col_idx*, *value*, and *node_info* from their respective BRAMs.
- **Weight Loader**: The *Weight Matrix* is initially stored in the **Weight BRAM**. To facilitate parallel computation, the **Weight Loader** assigns each column of the matrix to a separate BRAM. These column-specific BRAMs are then processed independently by dedicated **SP-PE** modules, enabling simultaneous computation of a single feature row across multiple weight columns.
- **SP-PE** (Sparse Processing Element): Each column-specific BRAM is processed by a separate **SP-PE** module. These modules receive non-zero values from the

Feature Loader and multiply them with the corresponding weights to compute a partial product for the matrix-vector multiplication.

- **WH BRAM:** After completing the computation of the vector $W\vec{h}$ for a row, the resulting data is stored sequentially in the **WH BRAM**, ready for the next stage of processing.

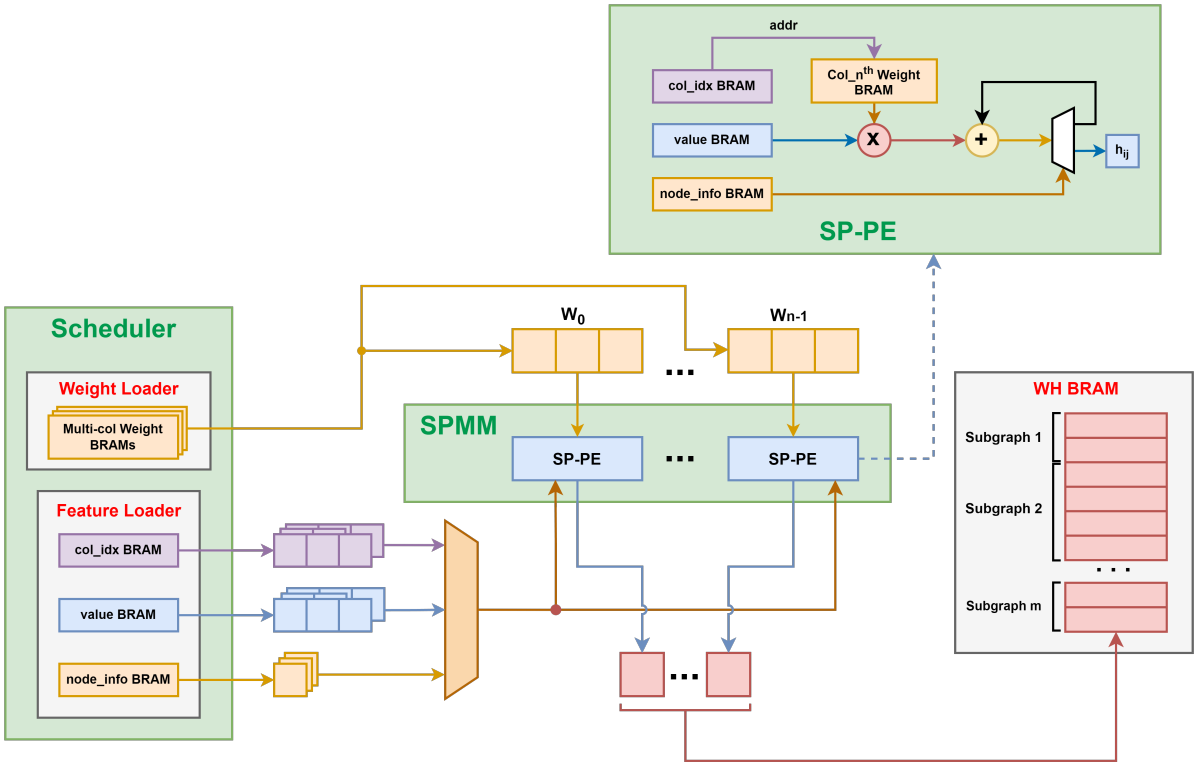


Figure 4.2: Hardware architecture of SPMM

SPMM operates in the following steps:

- In our design, the second layer contains 16 hidden channels. As a result, 16 **SP-PE** modules are instantiated, with each module responsible for processing one column-specific BRAM.
- The **Scheduler** coordinates the loading of features and weights. For each row in the feature matrix, the *node_info* is read from the **node_info BRAM**. This provides key details, including the number of non-zero values in the row (*row_length*), the total number of nodes in the subgraph (*num_of_nodes*), and whether the current node is a source node or a neighbor (*source_node_flag*).

- At each clock cycle, one non-zero value from H and its corresponding column index (*col_idx*) are fetched from their respective BRAMs. The *col_idx* is then used by each **SP-PE** to retrieve the corresponding weight value ($W_i[col\_idx]$) from its assigned column BRAM. Once retrieved, the non-zero value is multiplied with the weight ($value \times W_i[col\_idx]$), and this computation occurs concurrently across all 16 **SP-PE** modules. This parallel processing ensures that each **SP-PE** contributes to the partial computation of the $W\vec{h}$ vector for the current feature row. As the fetching and computation proceed in lockstep, the entire $W\vec{h}$ vector is completed once all non-zero values of the feature row have been processed, achieving both efficiency and without idle time across the modules.

- After computing the product for each non-zero value in the row, the partial results are accumulated. A counter keeps track of the completed calculations and compares them to *row_length*. Once the counter reaches *row_length*, indicating that all calculations for the row are complete, the *pe_ready_o* signal is asserted. The counter is then reset, enabling the system to start processing the next feature row.

- This also activates the *WH_BRAM_en* signal, which writes the resulting vector $W\vec{h}$ into the **WH BRAM**, concatenated with 2 information: *num_of_nodes* and *source_node_flag*, which will be used in the next computation steps.

4.3 DMVM

DMVM (Dense Matrix-Vector Multiplication) is responsible for performing the **Self-attention mechanism**. Each **WH** vector node is multiplied with the Attention Weight of either the source node or a neighbor node, based on the node's characteristics. The hardware implementation of the **DMVM** operation is represented by the equation below:

$$e_{ij} = \text{ReLU}(a_1 z_i + a_2 z_j) \tag{4.2}$$

Where:

- a_1 and a_2 represent the attention weights of the source node and neighbor node, respectively.

- z_i and z_j denote the $W\vec{h}$ vector of the source node and neighbor node, respectively.

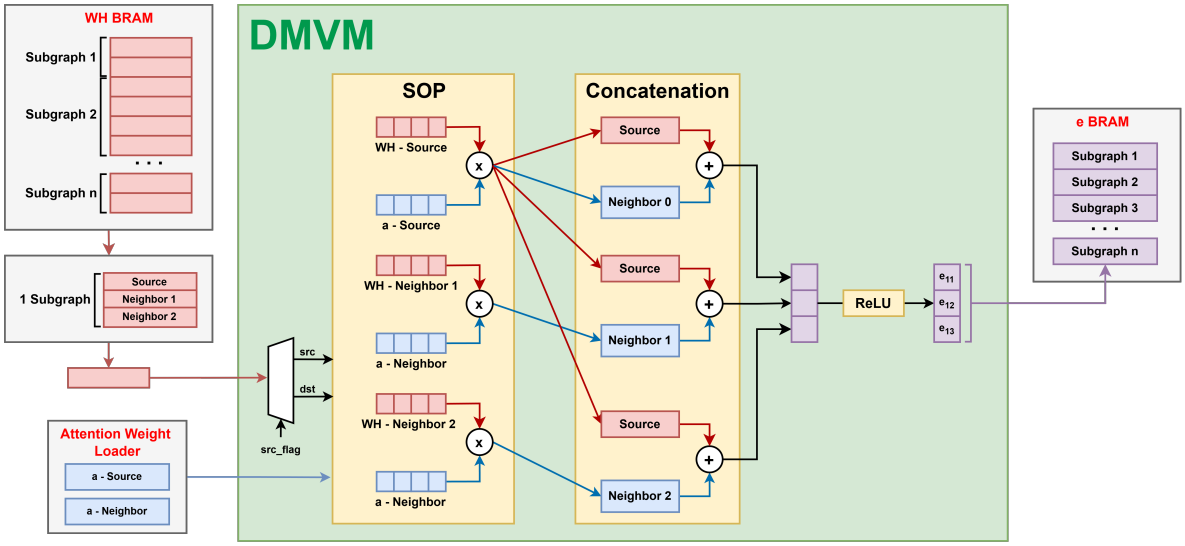


Figure 4.3: Hardware architecture of DMVM

In Figure 4.3, **DMVM** communicates with 3 components:

- **WH BRAM**: Each entry in the BRAM contains a $W\vec{h}$ vector, along with *num_of_nodes* and *source_node_flag*, which are stored after the **SPMM** phase.
- **Attention Weight Loader**: The *Attention Weight* vector, stored in the **a BRAM** by the **Scheduler** is split into 2 specific vectors stored in 2 registers: the first half corresponds to the *Attention Weight* for the source node, and the second half is for the neighbor nodes.
- **e BRAM**: After calculating all the attention coefficients e for a subgraph, these coefficients are stored in **e BRAM** for further processing.

DMVM operates as follows:

- The first $W\vec{h}$ vector in each subgraph represents the source node, while the remaining vectors correspond to neighbor nodes, as indicated by *source_node_flag*. Based on this flag, the $W\vec{h}$ vector is either multiplied with the *Attention Weight* vector of the source node or that of a neighbor node prepared from **Scheduler**.
- **SOP** (Sum of Products) is responsible for computing the dot product between a $W\vec{h}$ vector and the corresponding *Attention Weight* vector. It consists of two consecutive phases:

– **Product:** In this phase, each pair of elements from the 2 vectors with the same index is multiplied in a single clock cycle, and the result is stored in a register.

– **Sum:** The next step is to calculate the sum of all products computed in the previous phase. To avoid critical path delays, the sum is computed iteratively rather than in a single cycle. The array of products is structured as a Adder Tree, where adjacent pairs of products are summed in parallel during each clock cycle. This reduces the number of items to be summed by half after each cycle. As a result, the total number of clock cycles required to compute the sum is approximately $log_2 N$, where N is the total number of products. Figure 4.4 illustrates an example of the sum calculation for 8 elements, which requires a total of 4 clock cycles.

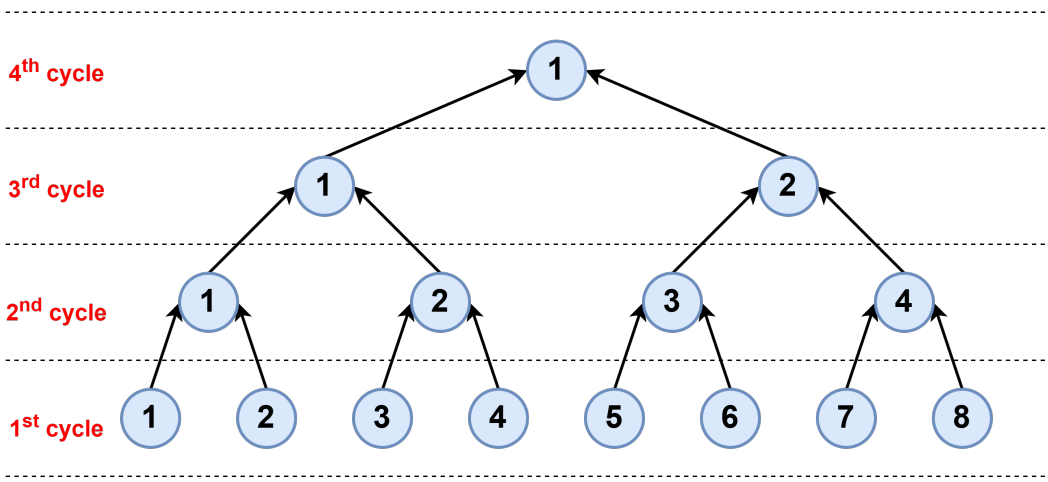


Figure 4.4: An example of sum calculation based on Adder Tree

- The process continues by reading the remaining $W\vec{h}$ vectors of the subgraph and passing them through the **SOP** phase. Once the number of vectors read from the **WH BRAM** reaches *num_of_nodes*, the system stops reading new data from the **WH BRAM** and proceeds to the **Concatenation** phase.

- In the **Concatenation** phase, after the **SOP** computation for each node in the subgraph is completed, all of these values are added to the **SOP** value of the *source* node. This operation performs the attention mechanism, allowing the source node to attend to itself as well as all its neighbor nodes.

- Finally, all the coefficients of a subgraph are passed to the Activation Function step. As proposed, we use **ReLU** to eliminate negative values, leaving only non-negative values as the final attention coefficients. These values are concatenated with *num_of_nodes* and written into the **e BRAM**, preparing for the **Softmax** process.

4.4 Softmax

Softmax is the module responsible for normalizing all attention coefficients e of a subgraph which are computed from **DMVM**, to effectively scale and highlight the importance of neighboring nodes.

$$\alpha_{ij} \approx \frac{2^{e_{ij}}}{\sum_{k \in \mathcal{N}_i} 2^{e_{ik}}}. \tag{4.3}$$

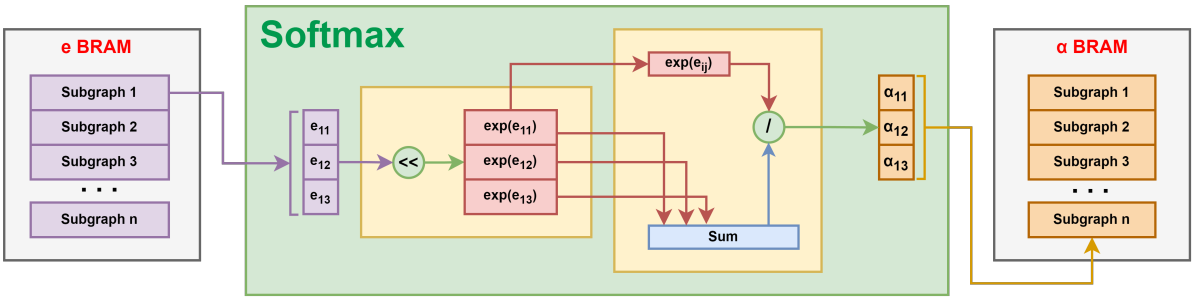


Figure 4.5: Hardware architecture of Softmax

From Figure 4.5, **Softmax** interacts with 2 BRAMS:

- **e BRAM**: Each data item in the **e BRAM** stores the attention coefficients e for all nodes in a subgraph, along with the *num_of_nodes*, which provides the total number of attention coefficients (and thus the number of nodes) in the subgraph.
- α **BRAM**: After normalizing all attention coefficients using the Softmax algorithm, the resulting coefficients α are concatenated into a single data item and written into the α **BRAM**, where they are used for processing in the final **Aggregator** equation.

Softmax operates in the following steps:

- Each data item in the **e BRAM** represents the attention coefficient for a node in the subgraph. The **Softmax** operation is applied to each data item sequentially.

- As proposed, we modify the original **Softmax** algorithm by replacing the e^x exponential function with 2^x. Since 2^x can be efficiently computed using bit-shifting operations in hardware, it requires only one clock cycle to calculate the base-2 exponential of all attention coefficients retrieved from **e BRAM**. However, the larger the range of the attention coefficients, the more hardware resources are required for the computation. To optimize resource usage, we choose to represent the attention coefficients e as signed 8-bit values, which range from -127 to 127. This allows the base-2 exponential to be computed within a total of 128 bits.

- After calculating the base-2 exponentials of all attention coefficients, we calculate their sum. To do this efficiently, we apply the Adder Tree technique, which reduces the total number of additions required. The sum is then stored in a register for the subsequent division step.

- Finally, each base-2 exponential of the attention coefficients is divided by the total sum, and the resulting value is stored in a register as a fixed-point representation.

4.5 Aggregator

The **Aggregator** is responsible for combining the outputs of the **DMVM** and **Softmax** modules to compute the final feature vector for each node. It multiplies the normalized attention coefficients from **Softmax** with the corresponding **WH** vectors from **DMVM**, accumulates the results, and applies the **ReLU** activation function. This process ensures that the computed feature vectors are prepared for subsequent layer.

$$h_i^{'} = ReLU(\sum_{j \in \mathcal{N}_i} \alpha_{ij} z_j) \tag{4.4}$$

The **Aggregator** is connected to three BRAMs:

- **WH BRAM**: It includes an additional read port dedicated to the **Aggregator**, allowing it to read and process the $W\vec{h}$ vector independently from the **DMVM** module.

- α **BRAM**: Each entry in α **BRAM** holds an array of all normalized attention coefficients for a subgraph, computed by the **Softmax** module.

- **Feature BRAM**: After the computation of a node's feature vector is complete, it is stored in the **Feature BRAM**, allowing the **PS** to read, preprocess, and prepare it for computation in the subsequent layer.

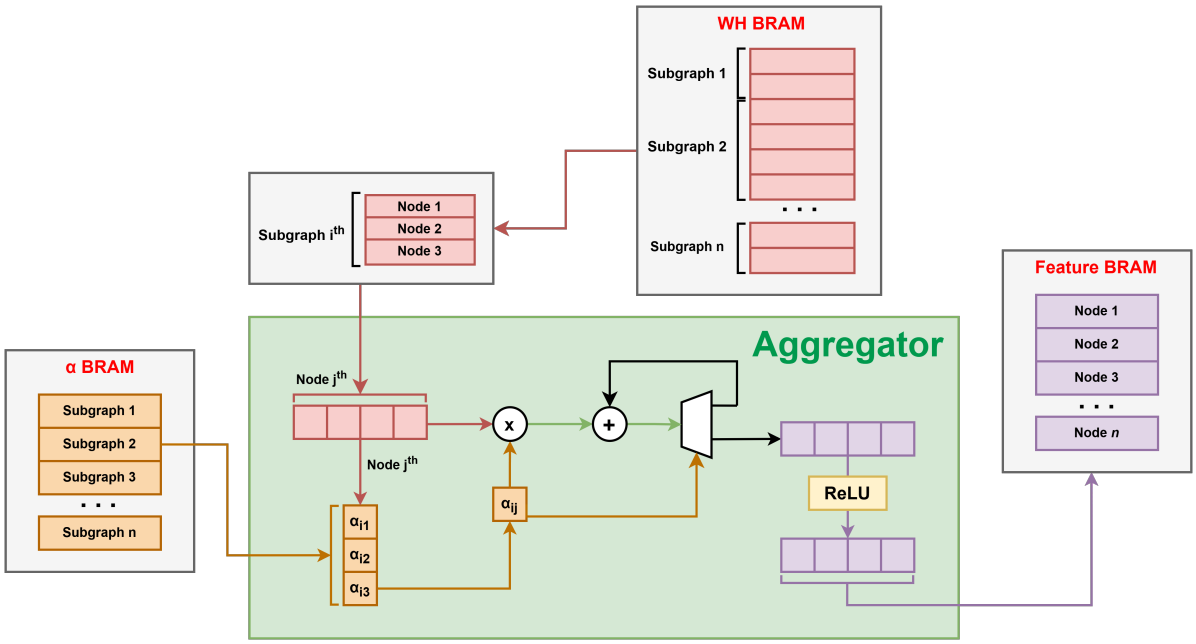


Figure 4.6: Hardware architecture of Aggregator.

The **Aggregator** operates as follows:

- To compute the feature vector for a node, the **Aggregator** multiplies the $W\vec{h}$ vectors with the normalized attention coefficients of the source node and its neighbor nodes. The results are accumulated and then passed through the **ReLU** activation function to finalize the computation.

- First, a list of all normalized attention coefficients for one subgraph is fetched from the α **BRAM**. Simultaneously, each $W\vec{h}$ vector is fetched consecutively from the **WH BRAM**, multiplied with its corresponding attention coefficient, and accumulated progressively.

- Once all the $W\vec{h}$ vectors for the subgraph have been processed, the accumulated results are passed through the **ReLU** activation function. This step removes negative values, retains non-negative values, and significantly simplifies computations for the next layer.

- Finally, the resulting array is flattened and written to the **Feature BRAM**, preparing it as input for the subsequent layer.

Chapter 5

Implementation

In Figure 4.1, the system is described and not specific to any SoC or FPGA device. For this thesis, the system is implemented on Xilinx SoCs, so several implementations are only compatible with Xilinx devices. The component which is heavily device-dependent is the communication bus. AXI interface protocol is used for Xilinx SoCs to implement the bridge between PS and PL. Xilinx AXI is a bus protocol for on-chip communication between IP cores in FPGAs and SoCs. It provides a standard interface for high-performance data transfer between devices while minimizing interconnects. The protocol is widely used in Xilinx FPGAs and SoCs, as well as in many other digital systems. Besides the AXI interface, the system also needs several IPs to support the communication, as shown in Figure 5.4.

5.1 System Implementation

5.1.1 Processing System

The Zynq Processing System (PS) is a pre-designed IP block that integrates a dual-core ARM Cortex-A9 processor, programmable logic (PL), and on-chip peripherals. It simplifies the design process by providing a complete system that can be customized, enabling efficient development of system-on-chip (SoC) solutions without starting from scratch.

This IP provides two communication ports between the PS and PL: **S_AXI_HP0_FPD** and **M_AXI_HPM0_FPD**. S_AXI_HPx_FPD is a high-performance port designed for

connecting the PL and DRAM, offering high-bandwidth access to external memory and peripherals. S_AXI_HPMx_FPD is used to map memory spaces, such as weights and feature input/output maps, between the PS DRAM and PL memory. On the other hand, M_AXI_HPMx is a high-speed interface for transferring large volumes of data from the PS to the PL. Figure 5.2 shows the interface list the IP, which all of the interfaces have maximum bit width of 128.

Figure 5.1: Zynq IP in Vivado with interfaces

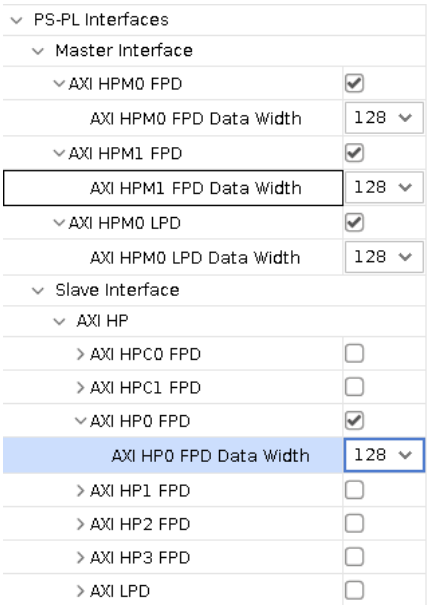


Figure 5.2: Interfaces configuration

The **pl_clk0** port in the PS manages the system reset and connects to the IP Processor System Reset, allowing designers to control the reset functionality (active high or low) of different system components. This IP block enhances system flexibility and performance, making it ideal for FPGA-based SoC designs.

5.1.2 PS - PL Communication

Communication between PS and PL in a heterogeneous SoC like MPSoC requires protocols and modules for high-speed data transfer. The GAT model involves large and complex data formats, so BRAM is used under PL for storage and computation. To transfer data from PS to PL, DMA (Direct Memory Access) is employed, enabling fast data transfer between PS and PL memory without CPU intervention, optimizing bandwidth and reducing latency.

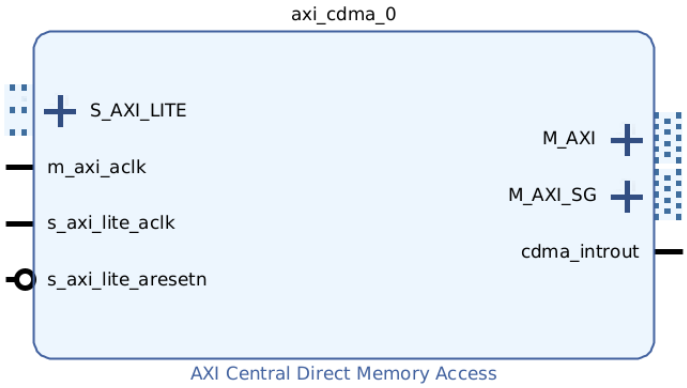


Figure 5.3: CDMA IP in Vivado

In our approach, instead of storing all data in one memory block, the communication method uses CDMA with multiple BRAM Controllers to store data, dividing it into manageable parts for parallel or distributed processing in the PL. Each BRAM Controller has its own access address. If the addresses are discrete (non-contiguous), the processor must execute multiple transfer operations, specifying each destination address separately, which increases the overhead and slows down the transfer process. However, by configuring the addresses to be contiguous, the PS can send all the data in a single transfer operation to the BRAMs. CDMA ensures that the data is distributed correctly across the BRAMs, simplifying implementation and reducing communication time between the PS and PL.

To enable this efficient data transfer, the **AXI Interconnect** connects the Zynq PS to CDMA, facilitating communication between the Processing System and the Programmable Logic. Meanwhile, **AXI SmartConnect** is used to link CDMA with multiple BRAM Controllers, ensuring optimal data routing when there are multiple BRAMs involved. By utilizing these components, the system ensures efficient data flow with reduced latency, enabling seamless interaction between the different parts of the design.

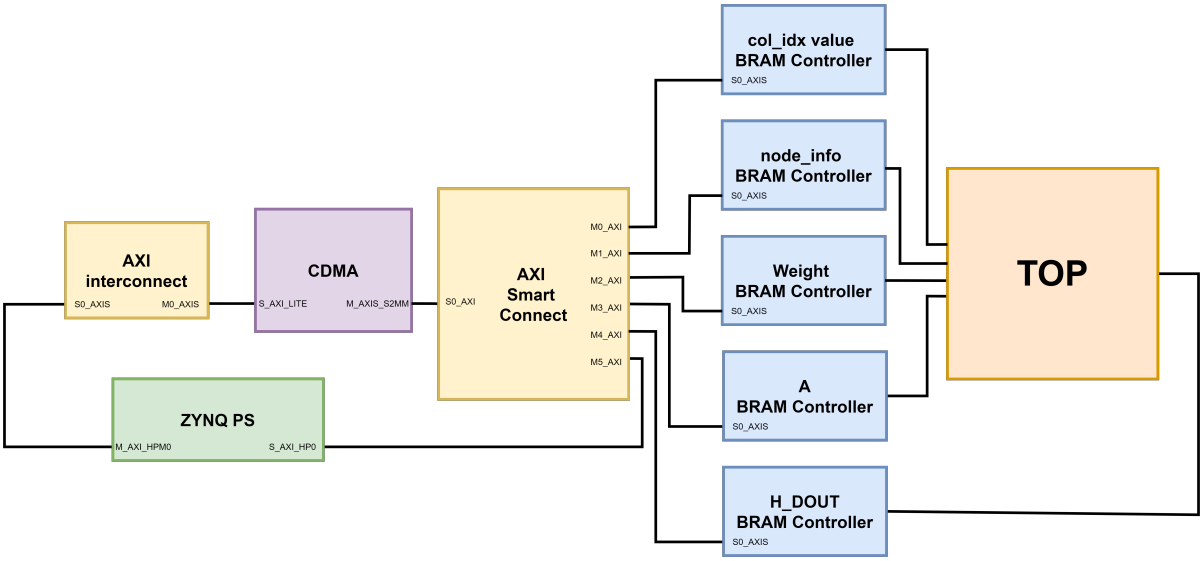


Figure 5.4: Communication between PS and PL using CDMA with multiple BRAM Controllers

5.2 Software Implementation

5.2.1 GAT Training

Dataset

Cora Dataset

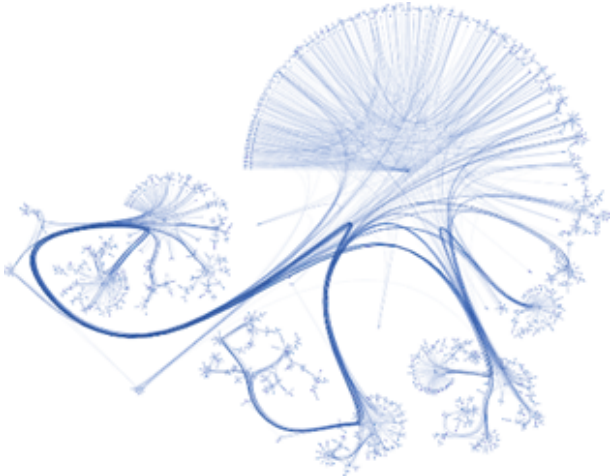


Figure 5.5: yEd Live Visualization Cora Dataset

- **Type**: Citation Network.

- **Description**: The Cora dataset is one of the most popular citation network datasets for evaluating machine learning algorithms in graph-based settings. It consists of scientific papers (nodes) from various fields of computer science, which are connected through citation relationships (edges). Each paper is represented as a node in the graph with a bag-of-words feature vector that describes its content. The Cora dataset is often used for node classification tasks, where the goal is to predict the topic of each paper based on its content and citation relationships. The nodes are classified into one of 7 categories of computer science research, such as neural networks, machine learning, and data mining,…..
- **Nodes**: Represent documents (research papers) **(2708 nodes)**.
- **Edges**: Represent citations between papers **(10556 edges)**.
- **Features**: Each node has a bag-of-words feature vector **(a 1433-dimensional binary vector)**.
- **Classes**: Correspond to different research topics (e.g., neural networks, machine learning, data mining, etc.) **(7 classes)**.
- **Task**: Node classification (assign each node a label from one of the 7 classes).

CiteSeer & PubMed Dataset

(The structure is similar to Cora dataset, with a different number of nodes, edges, classes,...)

Training models

Graph Attention Layer: The two GAT layers in the model work together to process graph data and learn meaningful representations for each node. The first layer aggregates information from neighboring nodes using attention mechanisms, creating enriched node embeddings that capture relationships in the graph. The second layer further processes these embeddings, transforming them into output predictions that represent the final classifications for each node. This combination of layers allows the model to effectively integrate local and global graph information for accurate prediction

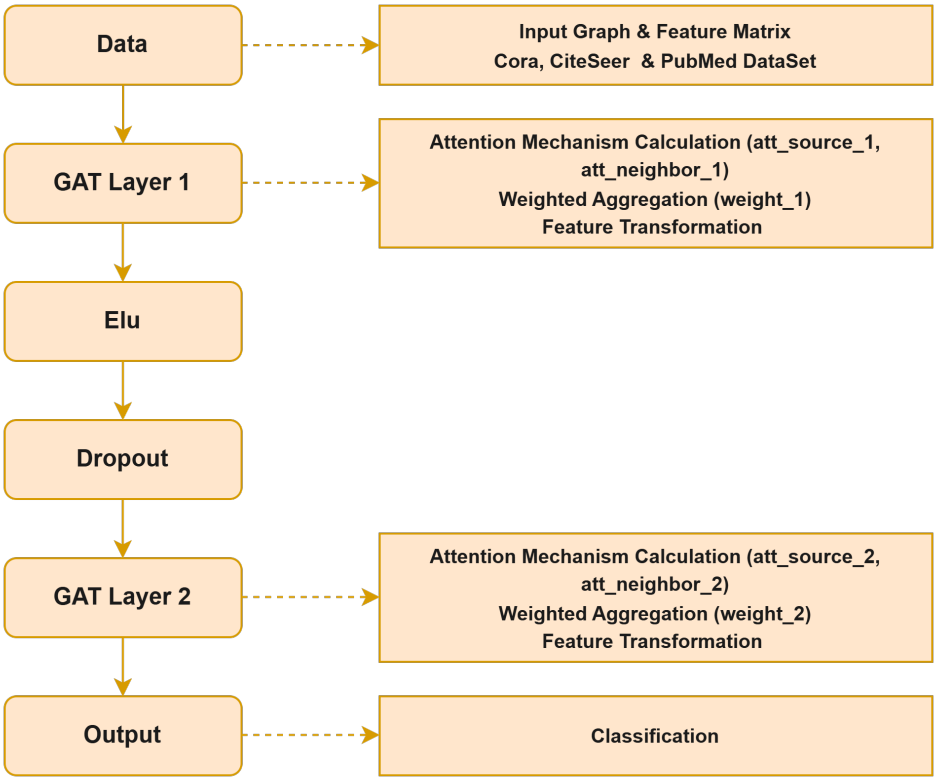


Figure 5.6: Two Layer GAT Training Model

Dropout: Dropout is applied to the output of first layer (GAT Layer 1) with a probability of 0.6, meaning 60% of the values are randomly set to zero during training. Dropout acts as a regularization technique by introducing noise into the input data, forcing the model to rely on multiple features rather than overemphasizing specific ones. This randomization encourages the model to learn more **robust** and **generalizable** patterns, as it cannot depend solely on any single input feature during training. By **reducing the risk of overfitting**, dropout ensures that the model performs **better on unseen data** and enhances its overall ability to generalize **across different datasets**.

Elu: After the first GAT layer, the Elu function is applied to the layer's output to introduce **non-linearity** into the mode. This behavior enables the model to **avoid "dead neurons"** a problem where certain nodes may output zero gradients, effectively halting their contribution to the learning process. Unlike simpler activation functions such as ReLU, ELU has a **smoother curve for negative inputs** , which helps maintain **small but non-zero gradients** during backpropagation.

Implementation through GAT class

```python
class GAT(torch.nn.Module):
    def __init__(self,
                 hidden_channels = Configuration["GAT"]["hiddenChannel"],
                 heads = Configuration["GAT"]["head"]):
        super().__init__()
        torch.manual_seed(1234567)
        self.conv1 = GATConv(dataset.num_features, hidden_channels,heads, True)
        self.conv2 = GATConv(heads*hidden_channels, dataset.num_classes,1, False)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.elu(x)
        x = F.dropout(x, p=0.6, training=self.training)
        x = self.conv2(x, edge_index)
        return x
```

Figure 5.7: GAT Class Implementation

5.2.2 Quantization

Implementation through BuildModel class

```python
class BuildModel():
    ......
    def single_train(self):
        a = self.model.state_dict()
        for k, v in a.items():
            # Quantized
            scaled_tensor, dequantized = quantized(v,
                                                   Configuration["BuildModel"]["scaleMin"],
                                                   Configuration["BuildModel"]["scaleMax"],
                                                   torch.int8)

            # Dequantized
            converted_tensor = dequantized(scaled_tensor)
            a[k] = converted_tensor
        self.model.load_state_dict(a);
    ......
```

Figure 5.8: BuildModel Class Implementation

During this phrase, we apply **self-aware quantization (QAT)** to optimize the GAT

model by reducing 32-bit floating-point computations to 8-bit integers. During training, "fake quantization" (quantization & dequantization process in sequence) is used to simulate the effects of reduced precision, allowing the model to adapt and maintain accuracy. Observers are employed to monitor activation ranges, ensuring proper scaling factors are computed. The process involves quantizing weights and activations to 8 bits for computation and then dequantizing back to higher precision for gradient updates.

5.2.3 Graph Data Format

GCSR: In GAT, the feature matrix is typically **sparse**, and to optimize storage and computational efficiency, it is compressed to retain only **nonzero elements**. To support graph processing, the **GCSR** format is introduced, representing graph data through three arrays: **col-index**, storing column indices of nonzero elements; **value**, storing the nonzero values; and **node-info**, which tracks the number of nonzero elements per row **(row_length)**, number of nodes of each subgraph **(num_of_nodes)**, and identifies whether a node is a source or neighbor **(node_flag)**. Each row of the adjacency matrix is treated as a subgraph, capturing interactions between a node and its neighbors, with overlaps to prevent boundary effects.

Implementation through GCSR_Data_Compression_Builder class

```
class GCSR_Data_Compression_Builder():
  def getInfoGCSR(self, printInfoShape = Configuration["GCSR"]["printInfoShape"]):
    ...
        subgraph_info_gcsr[int(node_idx)] = {
            'source_node_nonzero_indices': ...,
            'neighbors_nonzero_indices': {....},
            'num_of_nodes': ...
        }
    ...
```

Figure 5.9: GCSR – Data Compression Builder

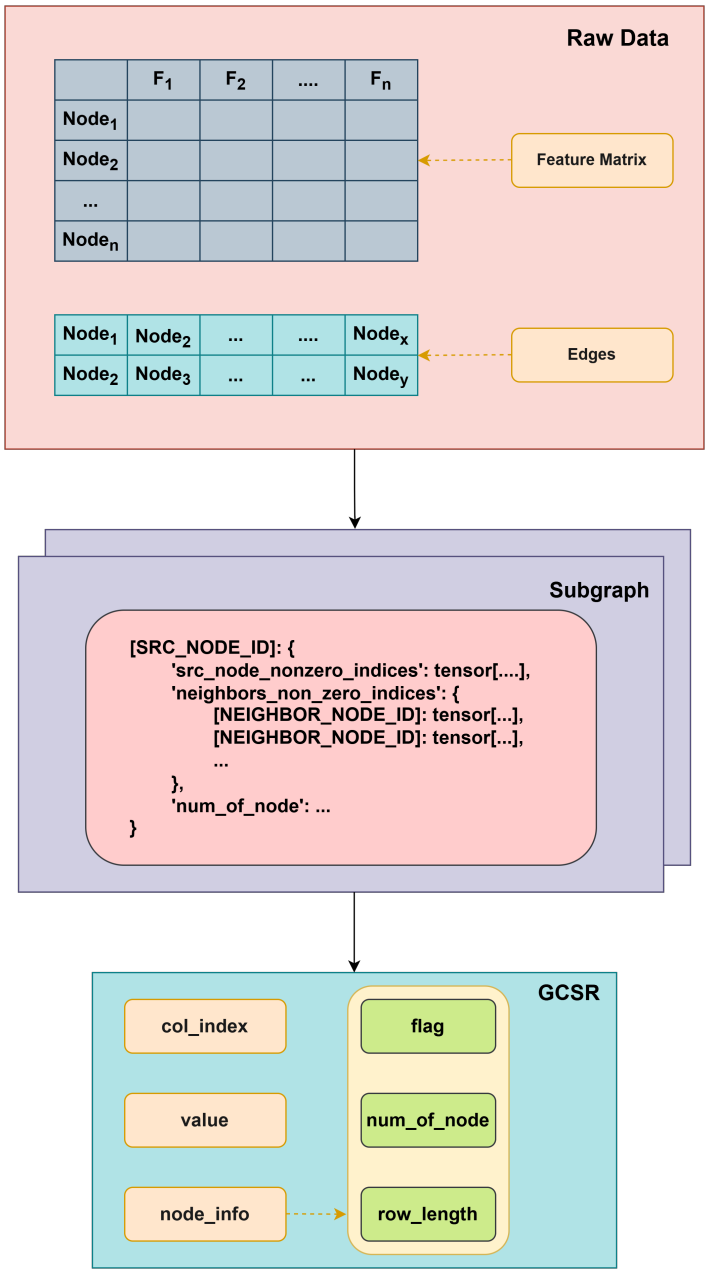


Figure 5.10: GCSR – Data Compression Builder

5.3 Hardware Implementation

5.3.1 Hardware Acceleration of Multiplication Operation

In hardware design, multiplication is a fundamental operation, but it can introduce significant delays if not carefully implemented. Typically, a simple multiplication between two numbers a and b in hardware can be expressed as:

```
1    assign p = a * b;
```

However, the traditional approach to multiplication in hardware, especially for more complex operations involving large numbers, can lead to significant delays. This is because multiplication usually involves multiple stages of computation, which can introduce critical path delays and result in long latency, particularly when combinational logic is added. This can severely impact the system's ability to operate at high frequencies, limiting its performance.

To address this issue, we focus on optimizing the multiplication operation by replacing the traditional multiplication approach with more efficient bitwise operations such as bit shifts and additions. These operations can be performed faster, with less critical path delay, enabling the system to run at a higher frequency. The key idea is to express multiplication as a series of bit shifts and additions based on the binary representation of the numbers involved.

Optimized Multiplication Using Bit Shifting and Add Operations

Every integer can be represented as a sum of powers of 2. This property allows multiplication to be replaced with a combination of shift and add operations, which can be computed much faster. Specifically, a binary number b can be represented as:

$$b = b_0.2^0 + b_1.2^1 + b_2 + 2^2 + ... + b_n.2^n \tag{5.1}$$

Where b_i is the i^{th} bit of the binary representation of b, taking the value of either 0 or 1. With this representation, the multiplication of a and b can be rewritten as:

$$a.b = a.(b_0.2^0 + b_1.2^1 + b_2 + 2^2 + ... + b_n.2^n) = \sum_{i=0}^{n} a.b_i.2^i \tag{5.2}$$

Hence, each bit b_i determines whether a particular shift and addition operation is required.

- If $b_i = 0$, the corresponding product term contributes 0 to the result.
- If $b_i = 1$, the corresponding product term becomes $a.2^i$, which can be computed by shifting a to the left by i positions.

Hardware Implementation of Optimized Multiplication

In hardware, we can implement this optimized multiplication using a combination of multiplexers and bit shifts. The following example demonstrates how to multiply two 8-bit numbers a and b by performing parallel shift and add operations:

```
1   logic [7:0]   a, b;
2   logic [15:0]  p;
3
4   assign p = (b[7] ? (a << 7) : 0) +
5              (b[6] ? (a << 6) : 0) +
6              (b[5] ? (a << 5) : 0) +
7              (b[4] ? (a << 4) : 0) +
8              (b[3] ? (a << 3) : 0) +
9              (b[2] ? (a << 2) : 0) +
10             (b[1] ? (a << 1) : 0) +
11             (b[0] ? (a << 0) : 0);
```

In this example:

- For each bit b_i of the multiplier b, if $b_i = 1$, the value a is shifted left by i positions (which is equivalent to multiplying a by 2^i).
- The results of each shifted operation are then added together to produce the final product.

Chapter 6

Experiment results

6.1 Model Training

6.1.1 Without Quantization

Visualize Cora Dataset without training model

(a) 2-D Visualization (No Training)

(b) 3-D Visualization (No Training)

Figure 6.1: Visualization of no-training data

Training GAT Model

Detailed Step:

The model will be trained using early stopping instead of a fixed epochs (to halt training when performance stagnates, preventing overfitting and improving efficiency). The **Adam optimizer** and **Cross-Entropy Loss function** are used to guide the training process. The **training** procedure consists of the following steps:

- Clear the gradients to ensure no residuals from previous steps.
- Perform a single forward pass to compute the model's predictions.
- Calculate the loss based on the nodes used for training
- Recalculate gradients and update the model parameters accordingly

For the **testing** phrase, the process includes:

- Predicting the classifications for the nodes.
- Checking how many nodes are classified correctly.
- Defining a variable to calculate the percentage of correctly classified nodes

Outcome:

Figure below shows that the results when training the GAT model are good, the accuracy is at about **0.825**

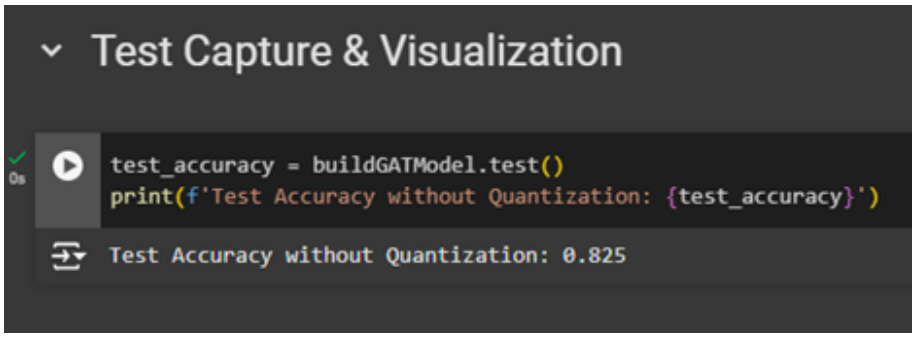
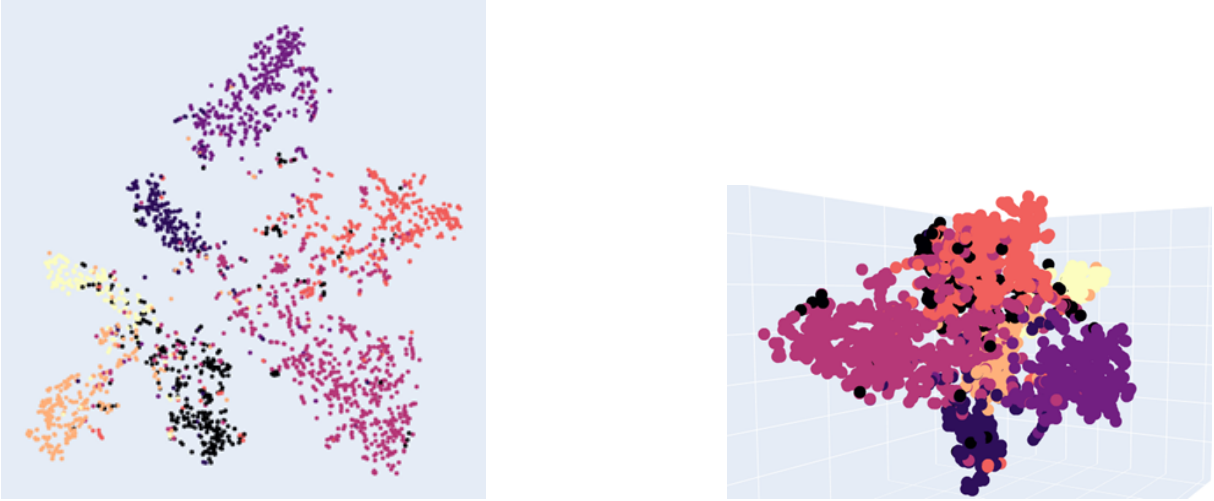


Figure 6.2: Accuracy after training model - without Quantization Aware Training

Visualization Result Classification



(a) 2-D Visualization (No Quantization)

(b) 3-D Visualization (No Quantization)

Figure 6.3: Visualization of result classification

6.1.2 With Quantization

Training GAT Model

Additional Step:

However, the model is currently being trained and computed using 32-bit floating-point numbers. To accelerate computation and reduce resource usage as mentioned in previous sections, **"Fake Quantization"** layers (Quantized and Dequantized Sequentially) will be added to quantize the model to 8-bit integers using the Quantization Aware Training method

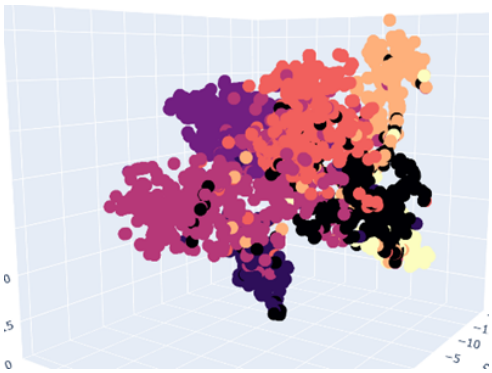
Outcome: Figure below shows that the results when training the GAT model when applied Quantization Aware Training (QAT) still maintain its accuracy around **0.822** (compared to **0.825**)

```
Test Capture & Visualization

test_accuracy = buildGATModel.test()
print(f'Test Accuracy with Quantization Aware Training (QAT): {test_accuracy}')

Test Accuracy with Quantization Aware Training (QAT): 0.822
```

Figure 6.4: Accuracy after training model - with Quantization Aware Training

Visualization Result Classification

(a) 2-D Visualization (Quantization Aware Training)



(b) 3-D Visualization (Quantization Aware Training)

Figure 6.5: Visualization of result classification

6.2 Simulation Results

6.2.1 Scheduler

6.2.2 SPMM

To verify the multiplication between H and W, we will test with the following matrices:

$$H = \begin{bmatrix} 11 & 0 & 0 & 0 & 3 \\ 0 & 3 & 0 & 0 & 1 \\ 2 & 9 & 4 & 0 & 0 \\ 0 & 13 & 8 & 0 & 5 \end{bmatrix} \quad \times \quad W = \begin{bmatrix} 24 & 14 & 7 & 3 & 28 & -28 \\ 14 & 21 & 16 & 2 & 2 & -29 \\ 20 & 9 & 22 & -9 & -22 & 19 \\ 16 & 13 & 18 & 13 & 2 & -3 \\ -26 & 20 & 31 & 11 & 1 & 2 \end{bmatrix}$$

The expected result z will be as follows:

$$z = \begin{bmatrix} 186 & 214 & 170 & 66 & 311 & -302 \\ 16 & 83 & 79 & 17 & 7 & -85 \\ 254 & 253 & 246 & -12 & -14 & -241 \\ 212 & 445 & 539 & 9 & -145 & -215 \end{bmatrix}$$

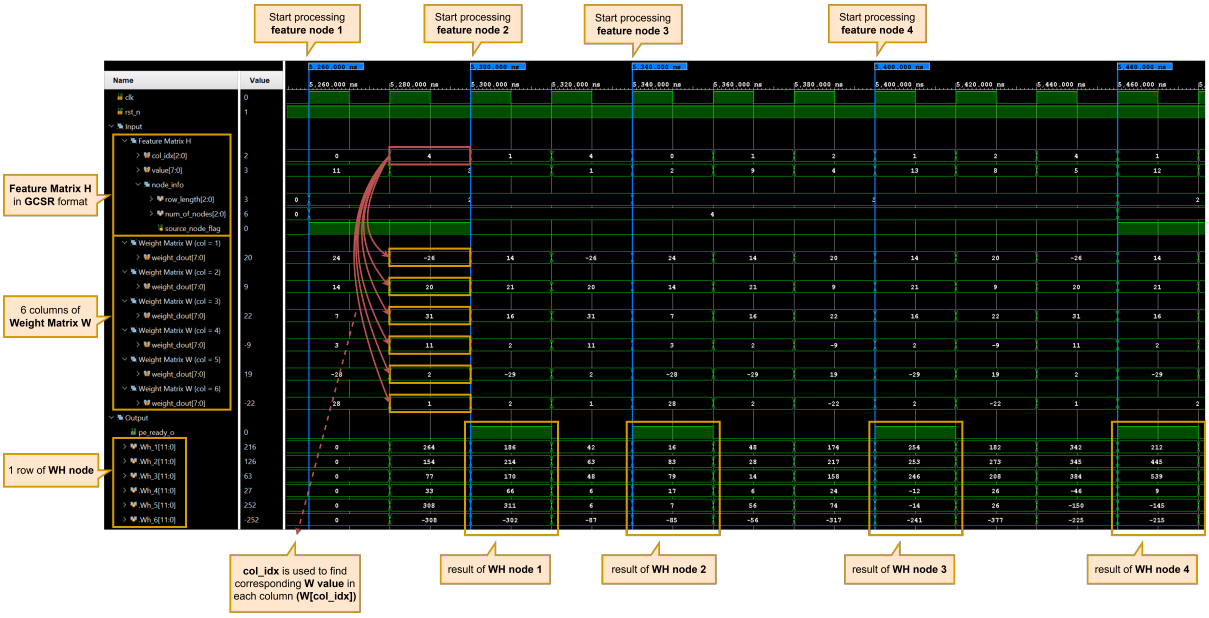


Figure 6.6: SPMM calculation simulation.

With the given **Feature Matrix** H as above, and following the GCSR format, it will be transmitted to **PL** in the following structure:

- **col_idx**: [0, 4, 1, 4, 0, 1, 2, 3, 1, 2, 4].
- **value**: [11, 3, 3, 1, 2, 9, 4, 13, 8, 5].
- **node_info**: [2||4||1, 2||4||0, 3||4||0, 3||4||0].

In **SPMM**, each pair of *col_idx* and *value* is fetched from the BRAM and sent to 6 **SP-PE** modules, each representing a column of the Weight Matrix W. The *col_idx* serves as the address to retrieve the corresponding weight from the column-specific BRAM in each **SP-PE** module, which is then multiplied with the *value*.

The resulting products are accumulated for the current row until all non-zero values are processed. This is tracked using a counter that increments with each fetched non-zero value. Once the counter reaches *row_length*, as specified by the **node_info BRAM**, it is reset, and the **pe_ready_o** signal is asserted. This signal indicates that the computation for the node is complete and triggers a write request to the **WH BRAM** to store the resulting vector for subsequent processing.

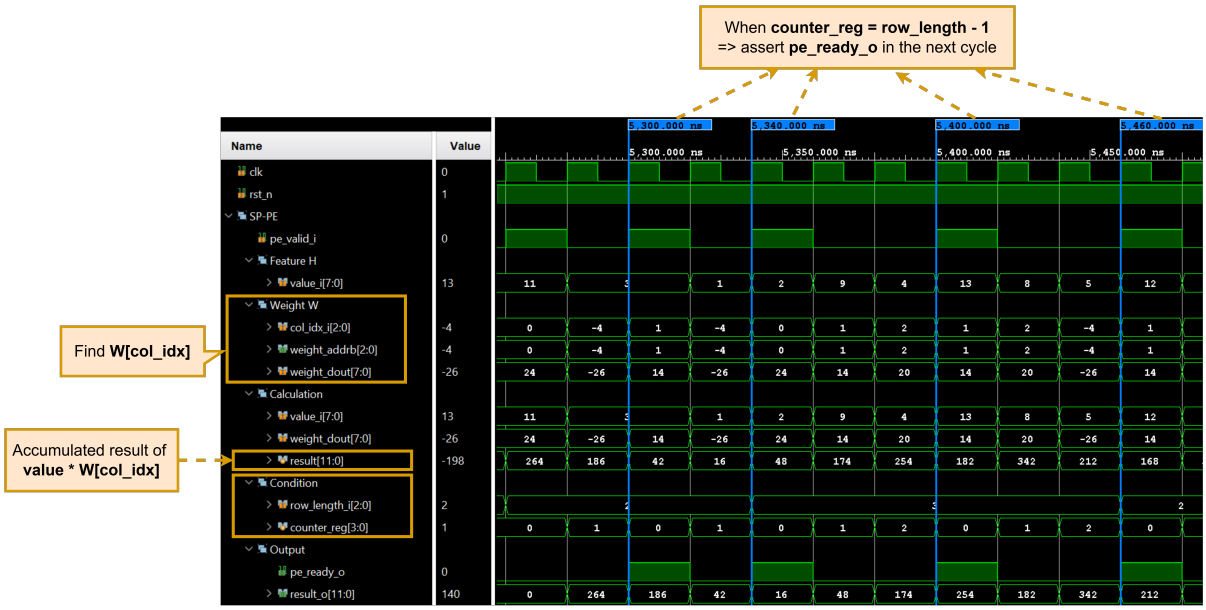


Figure 6.7: Calculation operation in one SP-PE module.

Chapter 7

Future Plan

7.1 Drawbacks

7.1.1 Our Models

The drawbacks include:

- **Dataset**: Currently, only a simple **Cora** dataset has been used to test the correctness as well as the methods for training, optimizing and quantization of the model. Moving forward, the focus will shift towards using more advanced datasets such as **Citeseer** and **PubMed**, which are widely recognized for their relevance in graph-related tasks. Additionally, the possibility of exploring other **practical** datasets will be considered to broaden the scope and applicability of the system. By incorporating these datasets, the aim is to test the system's capabilities in handling complex and real-world scenarios.

7.1.2 Our Design

The drawbacks include:

- **Feature Format**: Our current design is optimized for storing and processing the Feature Matrix H in the first layer of the Graph Attention Network, as the input Feature Matrix is highly sparse. To handle this efficiently, we proposed and applied the **GCSR** format to optimize the storage of H in hardware. However, after the

first layer is computed, the Feature Matrix becomes mostly dense. Continuing to use the same architecture for subsequent layers would be counterproductive, as it would consume more resources than storing the original dense matrix directly. Therefore, our current solution is we will adjust the storage and processing method for the H matrix (which will affect **SPMM** module) by storing the H matrix in its original dense format for subsequent layers.

- **Feature Pre-processing**: Another challenge arises after the Feature Matrix H is computed in the first layer, as it no longer retains the ”subgraph characteristic.” This is because each subgraph produces only a single feature node. Consequently, if the H matrix is used directly in the next layer, it cannot be processed with our proposed design and causes system idling (as previously explained). To address this, the H matrix must be pre-processed by concatenating all subgraphs into a unified structure. We are currently evaluating whether to implement this functionality directly in hardware or delegate it to the **PS** for processing.

- **DMVM**: In our proposed design for calculating the **self-attention mechanism**, the **DMVM** module does not currently support **pipelining**. This limitation arises because, for each $W\vec{h}$ vector sent to **DMVM**, several clock cycles are required to complete the dot product (including both multiplication and addition operations). To enhance system performance, we plan to redesign the **DMVM** module to support **pipelining**, enabling more efficient and faster processing.

7.2 Future Plan

7.2.1 Phase 1: RTL Coding

This phase focuses on implementing the remaining modules on hardware. Starting in the beginning of January, we will proceed with designing the DMVM, Softmax and Aggregator modules. Additionally, to verify accuracy and resource ultilization, we will implement the computations for the exponential function e^x and compare it with the approximation function 2^x. Once the RTL design for the first layer is released, we will move on to the second layer and continue to synthesis and modify the design. The phase

is expected to finish in the end of February.

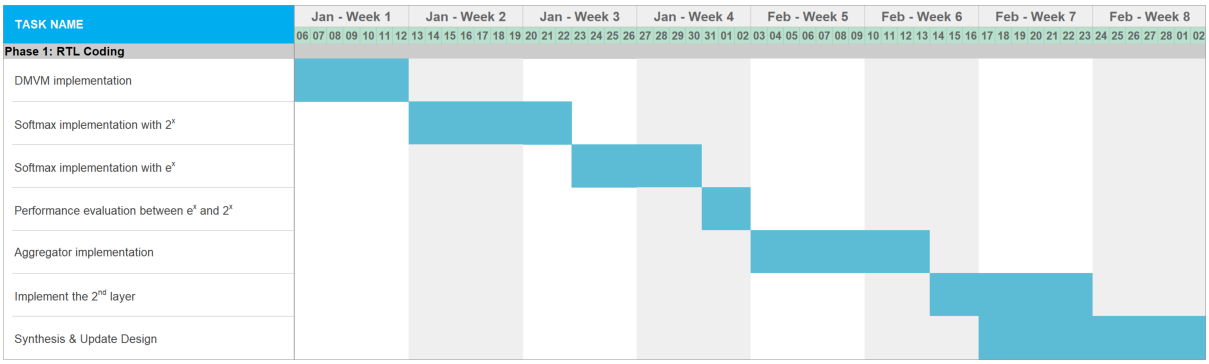


Figure 7.1: Phase 1 - RTL Coding

7.2.2 Phase 2: Training Data

The Training Data phase begins simultaneously with the RTL Coding phase (Phase 1) to prepare the input parameters and evaluate the model once the RTL Design is completed. This phase involves fine-tuning the model's parameters and training the model with additional datasets. The final step of this phase focuses on improving accuracy and assessing the model's overall quality.

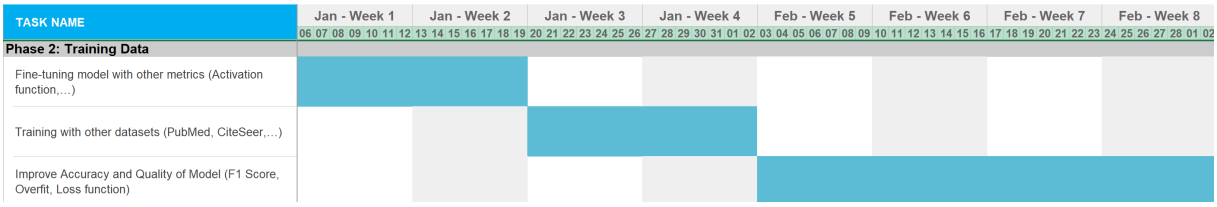


Figure 7.2: Phase 2 - Training Data

7.2.3 Phase 3: Simulation and Verification

This phase is executed in parallel, with some tasks running independently of RTL coding. Starting in January, we will explore simulation and verification methodologies, such as UVM, develop reasonable test plans for the systems, and construct a basic environment for verification. Once the environment is stable, we will proceed with standalone verification of individual modules (evaluating timing, signals, results, etc.) and block-level testing of the TOP module. This phase is planned to run continuously from Week 3 of January and is expected to conclude by the end of March.

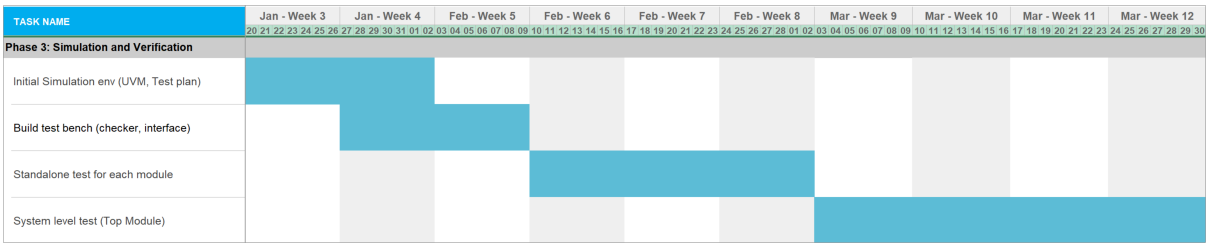


Figure 7.3: Phase 3 - Simulation and Verification

7.2.4 Phase 4: Implementation on FPGA

This phase involves working directly with the board. The first step is to review the MPSoC architecture and implement software code on the Processing System (PS). The team will deploy the training model directly onto a Jupyter Notebook supported by the board and utilize methods from the PYNQ framework to transfer data from the PS to the Programmable Logic (PL). Consequently, various experiments will be conducted, such as implementing a basic PL design and analyzing DMA transfer time. During this phase, a Block Design will be developed in Vivado, connecting the PS with CDMA and AXI Interconnect. The final step will involve utilizing datasets and transferring input parameters from the PL to the PS.

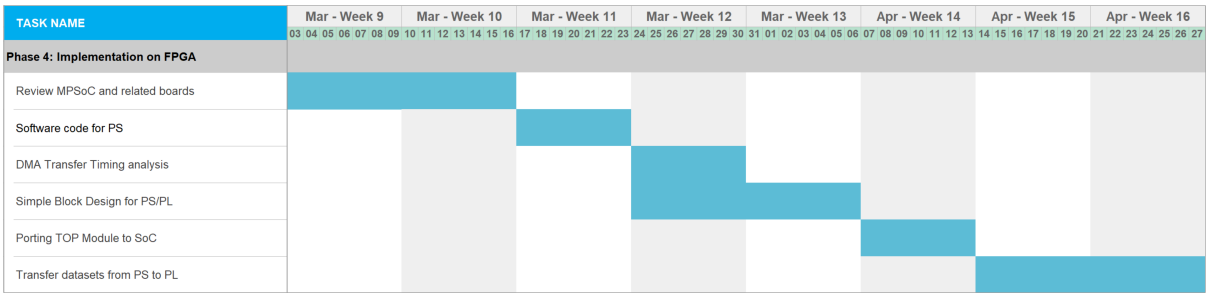


Figure 7.4: Phase 4 - Implementation on FPGA

7.2.5 Phase 5: Design Optimization

Finally, we will spend the last month optimizing our design. In four weeks, we will optimize frequency, utilize the hardware resource, as well as improve the accuracy of the model. We will conclude the overall system evaluation in Week 19, around May 18^{th}.

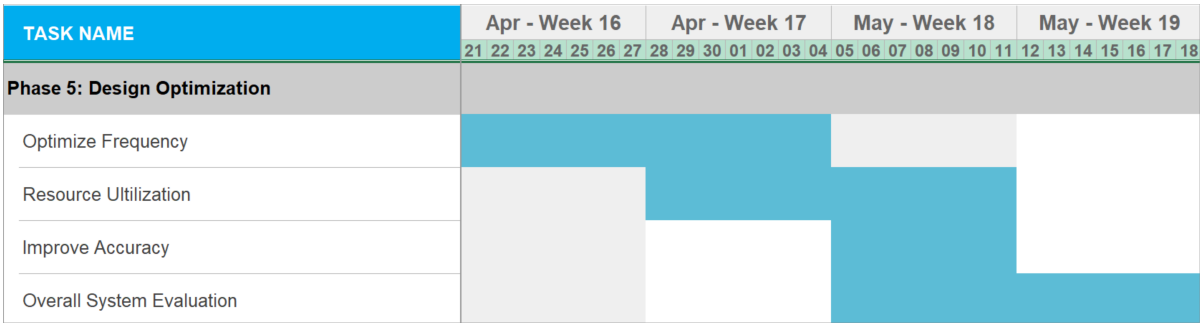


Figure 7.5: Phase 5 - Design Optimization

Chapter 8

Conclusion

This research has successfully proposed and partially implemented an architecture for accelerating Graph Attention Networks (GAT) on FPGA. Key achievements include fundamental studies on FPGA, SoC, and graph theory, training the GAT model with quantization to 8-bit integers, implementing the SPMM module, and exploring data transfer methods in Zynq MPSoC. However, challenges remain, such as addressing precision issues in 8-bit calculations, transitioning to fixed-point representation, and optimizing the Feature Matrix format for dense layers. Additionally, feature pre-processing and DMVM pipelining require further development to enhance system performance. Future work will focus on resolving these challenges, completing the remaining architecture, and evaluating performance on FPGA.

References

[1] Zerong He et al. “FTW-GAT: An FPGA-Based Accelerator for Graph Attention Networks With Ternary Weights”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 70.11 (Nov. 2023), pp. 4211–4215. DOI: 10.1109/TCSII.2023.3280180. URL: https://doi.org/10.1109/TCSII.2023.3280180.

[2] Can Lei, Huigang Wang, and Juan Lei. *SI-GAT: A method based on improved Graph Attention Network for sonar image classification*. 2022. arXiv: 2211.15133 [cs.CV]. URL: https://arxiv.org/abs/2211.15133.

[3] Jian Li, Yuwei Jian, and Yuxuan Xiong. “Text Classification Model Based on Graph Attention Networks and Adversarial Training”. In: *Applied Sciences* 14.11 (2024), p. 4906. DOI: 10.3390/app14114906. URL: https://doi.org/10.3390/app14114906.

[4] Nan Mu et al. “Graph Attention Networks for Neural Social Recommendation”. In: *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)* (2019), pp. 1320–1327. URL: https://api.semanticscholar.org/CorpusID:211206969.

[5] M. Procaccini, A. Sahebi, and R. Giorgi. “A survey of graph convolutional networks (GCNs) in FPGA-based accelerators”. In: *Journal of Big Data* 11 (2024), p. 163. DOI: 10.1186/s40537-024-01022-4. URL: https://doi.org/10.1186/s40537-024-01022-4.

[6] Zhuofu Tao et al. “LW-GCN: A Lightweight FPGA-based Graph Convolutional Network Accelerator”. In: *Journal of the ACM* 37.4 (Aug. 2018). 17 pages, 111:1–111:17. DOI: 10.1145/1122445.1122456. URL: https://doi.org/10.1145/1122445.1122456.

[7] Petar Veličković et al. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML]. URL: https://arxiv.org/abs/1710.10903.

[8] Renping Wang et al. "SH-GAT: Software-hardware co-design for accelerating graph attention networks on FPGA". In: *Electronic Research Archive* 32.4 (2024), pp. 2310–2322. ISSN: 2688-1594. DOI: 10.3934/era.2024105. URL: https://www.aimspress.com/article/doi/10.3934/era.2024105.

[9] Weian Yan, Weiqin Tong, and Xiaoli Zhi. "S-GAT: FPGA-Based Accelerator for Graph Attention Networks with High Performance and Energy Efficiency". In: (Dec. 2020), pp. 645–652. DOI: 10.1109/ICPADS51040.2020.00093. URL: https://doi.org/10.1109/ICPADS51040.2020.00093.

[10] Hang Zhao et al. "Multivariate Time-series Anomaly Detection via Graph Attention Network". In: (2020). Accepted by ICDM 2020, 10 pages. DOI: 10.48550/arXiv.2009.02040. arXiv: 2009.02040. URL: https://doi.org/10.48550/arXiv.2009.02040.