VIETNAM NATIONAL UNIVERISTY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY (HCMUT)
FACULTY OF COMPUTER SCIENCE & TECHNOLOGY

GRADUATION THESIS

ACCELERATION OF GRAPH ATTENTION NETWORK ON FPGA

MAJOR: COMPUTER ENGINEERING

THESIS COMMITTEE: CE-CC01

Supervisors:

Assoc. Prof. Dr. Pham Quoc Cuong - HCMUT

Reviewer:

Mr. Pham Kieu Nhat Anh - HCMUT

Authors:

Hoang Tien Duc - 2152520

Dang Hoang Gia - 2153312

Nguyen Duc Bao Huy - 2152089

HO CHI MINH CITY - June-2025

This thesis is dedicated for our parents and our instructors at HCMUT.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENT

We sincerely thank Associate Professor Pham Quoc Cuong for his kind and helpful support throughout our university studies and during the writing of this report. We are also thankful to Mr. Bui Anh Khoa, a K20 student, for taking the time to share his knowledge with us in our field of study.

We would like to express our gratitude to all the lecturers in the Faculty of Computer Science and Engineering at Ho Chi Minh City University of Technology. They have given us a strong foundation of knowledge, which was very important for our research and report.

To our families, we are deeply thankful for their emotional support and understanding, which allowed us to focus fully on this report. We are also grateful to our close group of friends in Computer Engineering for their support and encouragement throughout these four years. We hope each of us can achieve the goals we have worked so hard for.

Lastly, we want to thank ourselves for the effort and determination we put into this Computer Engineering project, even when we faced many challenges and misunderstandings.

Ho Chi Minh City, May 5^{th}, 2025
Hoang Tien Duc
Dang Hoang Gia
Nguyen Duc Bao Huy

ABSTRACT

Machine learning, with the advancement of artificial neural networks, has achieved significant milestones and is widely applied in various fields. However, as neural networks grow more complex, their computational demands increase, making it challenging to deploy them on resource-constrained devices, particularly for large models like Graph Attention Networks (GATs).

Graph Attention Networks (GATs) represent an advanced extension of Graph Neural Networks (GNNs), incorporating attention mechanisms to dynamically evaluate the importance of node connections. Despite their effectiveness, GATs are computationally intensive and resource-demanding, making deployment on resource-constrained devices challenging.

This project develops an FPGA-based architecture to accelerate GATs using techniques such as model quantization, efficient matrix operations, data transfer optimization, and parallel computation to improve performance and resource efficiency.

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
GOP	Gigabyte of Operations

1

INTRODUCTION

In this section we briefly introduce our thesis. This section includes an introduction to the context of this project, the research objective, the research scope, the research subject and the contents of this thesis.

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

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.

2

BACKGROUND AND RELATED WORK

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.

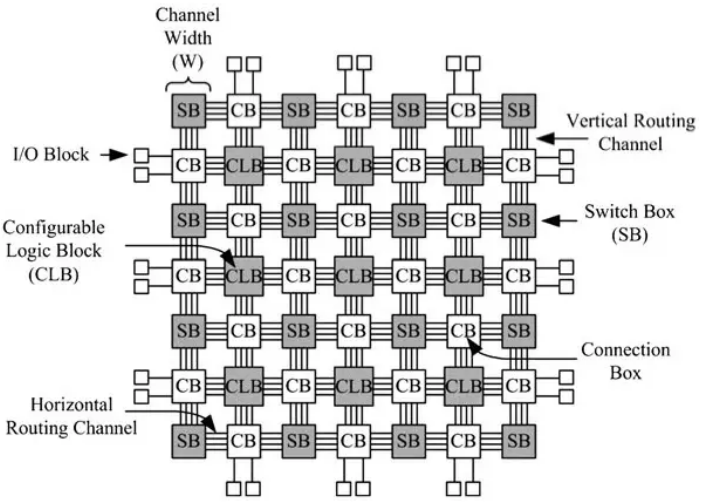


Figure 2.1: Components in FPGA

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.

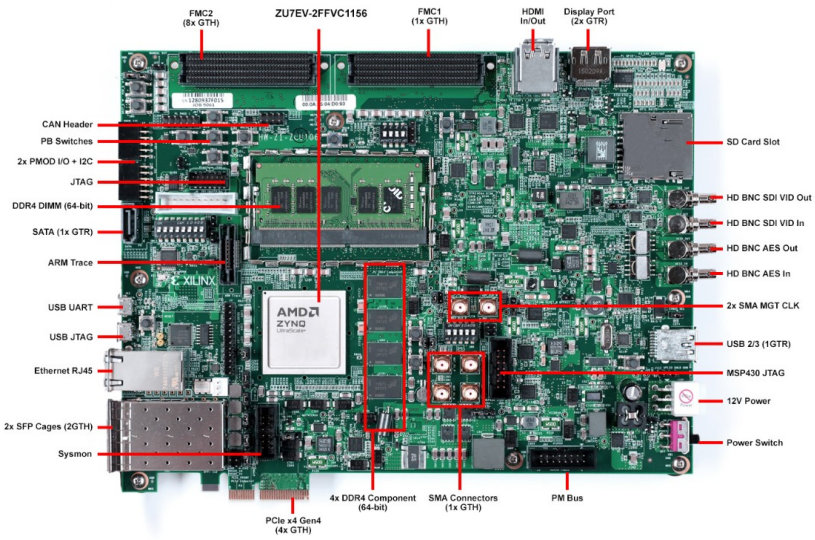


Figure 2.2: ZCU106

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.

Table 2.1: Resource on XCZU7EV

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

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.

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)

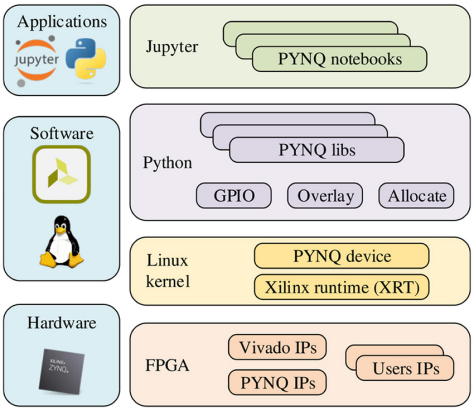


Figure 2.3: Pynq Framework

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,...

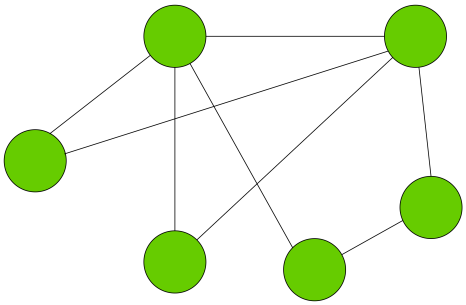


Figure 2.4: Node in graph

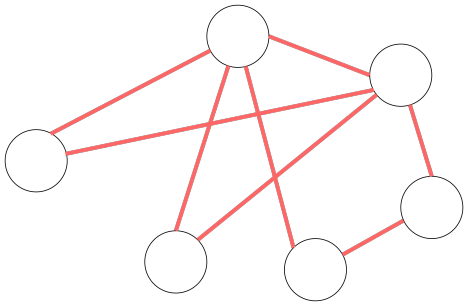


Figure 2.5: Edge in graph

Figure 2.6: Comparison of node attributes in three graphs

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.

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

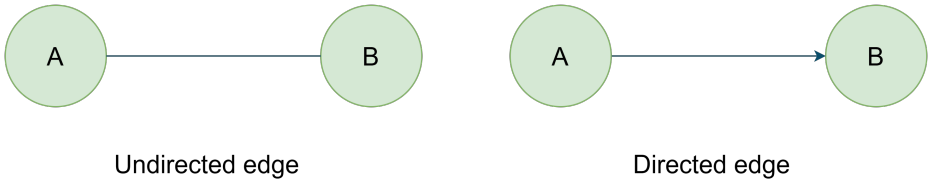


Figure 2.7: Edge representation

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.

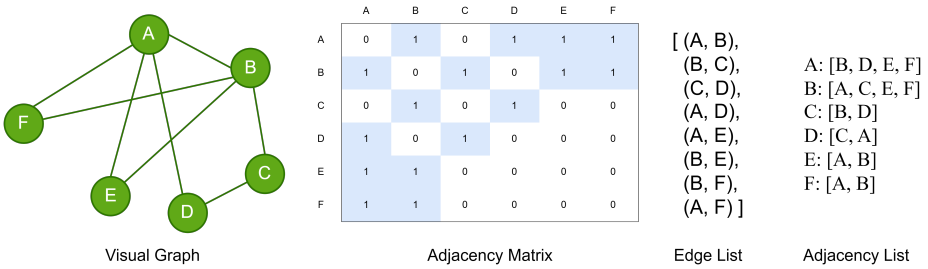


Figure 2.8: Representation of a Graph without Weights

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 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

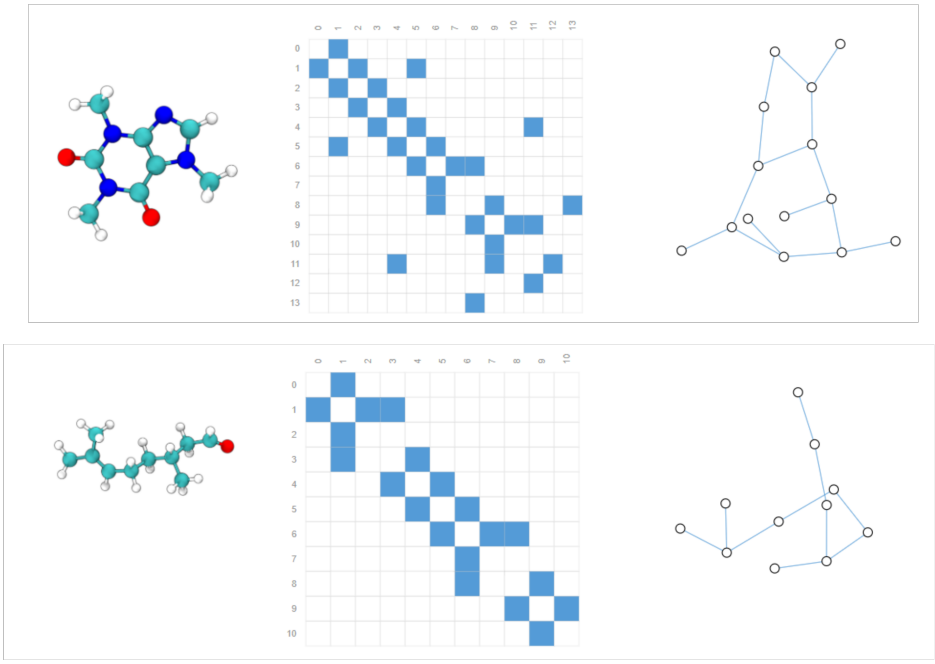


Figure 2.9: Graph can represent structural model such as molecule

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.

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**.

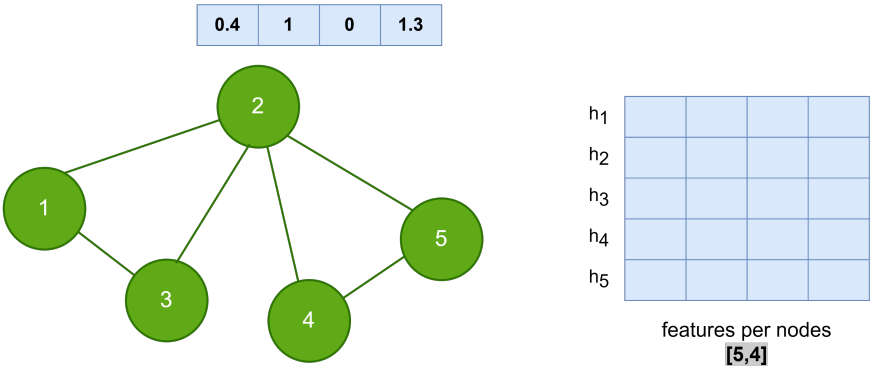


Figure 2.10: Features in Nodes

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

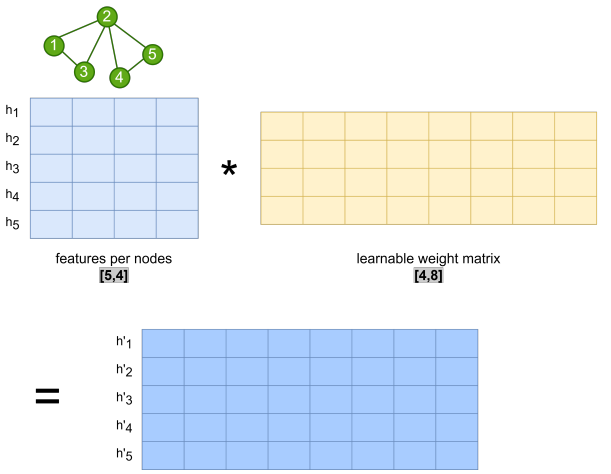


Figure 2.11: Matrix Multiplication represent Graph features

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.

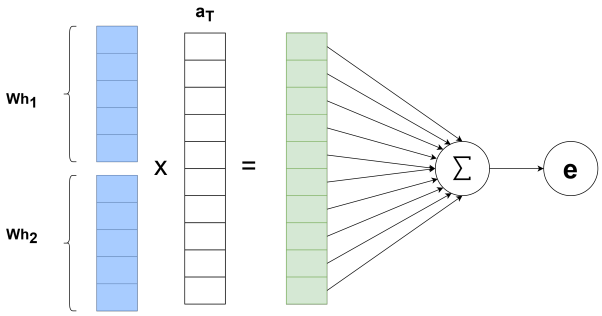


Figure 2.12: Calculation of Attention coefficient

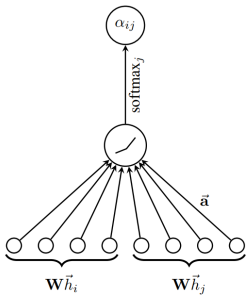


Figure 2.13: Softmax flow with LeakyRELU

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):

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

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,...)

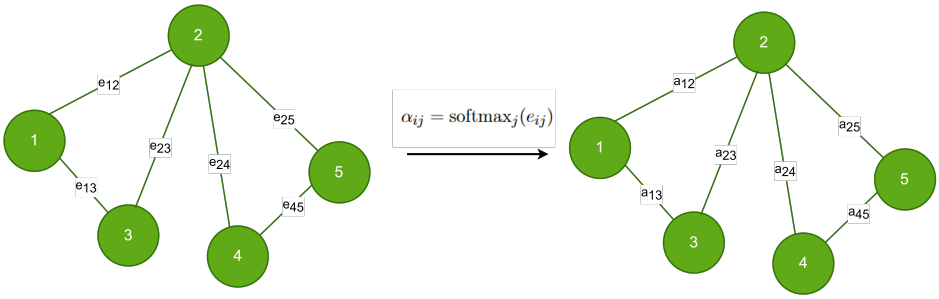


Figure 2.14: Attention Coefficient and Softmax

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):

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

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

2.6. RELATED WORK

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

To address the challenges in GNN acceleration, FP-GNN [6], an adaptive FPGA-based accelerator, has been proposed. FP-GNN introduces a unified processing module capable of simultaneously handling both Aggregation and Combination, alongside an Adaptive Graph Partition (AGP) strategy that alleviates memory bottlenecks and eliminates the need for graph repartitioning across layers. Moreover, multiple workflow optimizations are incorporated to balance workload and reduce feature sparsity. Implemented on a Xilinx VCU128 FPGA, FP-GNN demonstrates an average of 665× speedup and 3180× energy efficiency over CPU baselines, and 24.9× speedup and 138× energy efficiency over GPU baselines, achieving state-of-the-art performance compared to existing FPGA-based GNN accelerators.

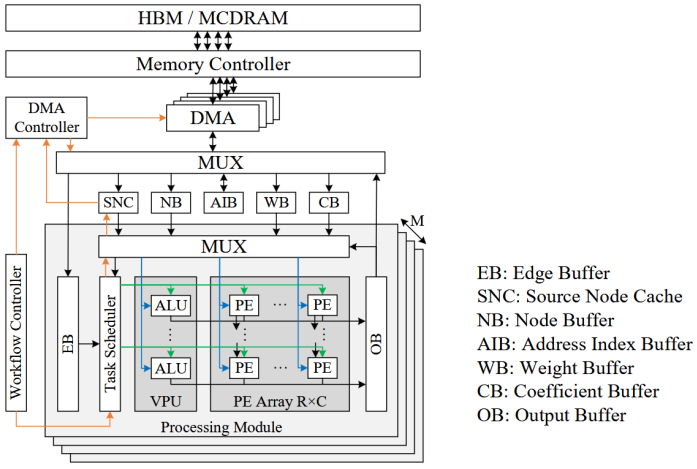


Figure 2.15: Architecture of FPGNN

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.

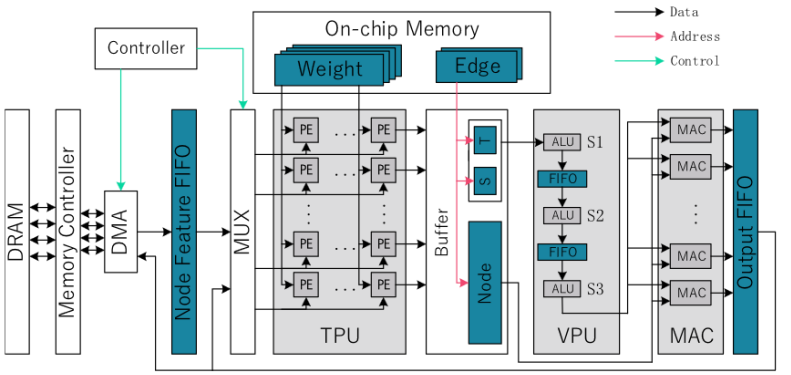


Figure 2.16: Architecture of FTW-GAT

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 parallelism 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.

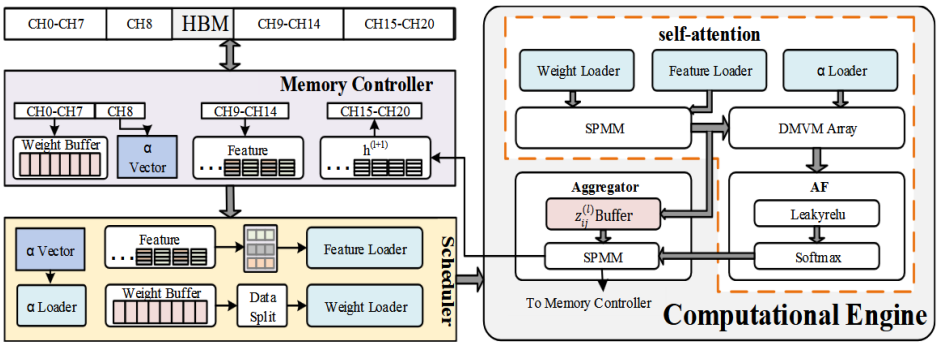


Figure 2.17: Architecture of SH-GAT

In general, Table 2.3 show the feature and performance of each Accelerator. All of the prior works used SOTA FPGA Board and achieve a high performance comparing to CPUs and GPUs.

Table 2.3: Comparison of FPGA-based GNN Accelerators

Accelerator	**Key Features**	**FPGA Board**	**Performance**
FP-GNN [6]	Unified processing for Aggregation and Combination, Adaptive Graph Partition (AGP), workflow optimizations for workload balance and sparsity elimination	Xilinx VCU128	665× CPU speedup, 3180× CPU energy efficiency; 24.9× GPU speedup, 138× GPU energy efficiency
FTW-GAT [1]	Ternary quantization, operation fusion, multi-level pipelining, graph partitioning	Xilinx VCU128	390× CPU speedup, 17× GPU speedup, 4007× CPU energy efficiency
SH-GAT [8]	Weight splitting, load-balanced SPMM, SoftMax approximation, prefetching	Xilinx Alveo U280	Up to 3283× CPU speedup, 13× GPU speedup, 2.3× prior FPGA accelerator

To address the limitations of prior works, our design tries to enhance computational parallelism and accelerate inference while optimizing storage resource utilization.

3

METHODOLOGICAL APPROACH

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.

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.

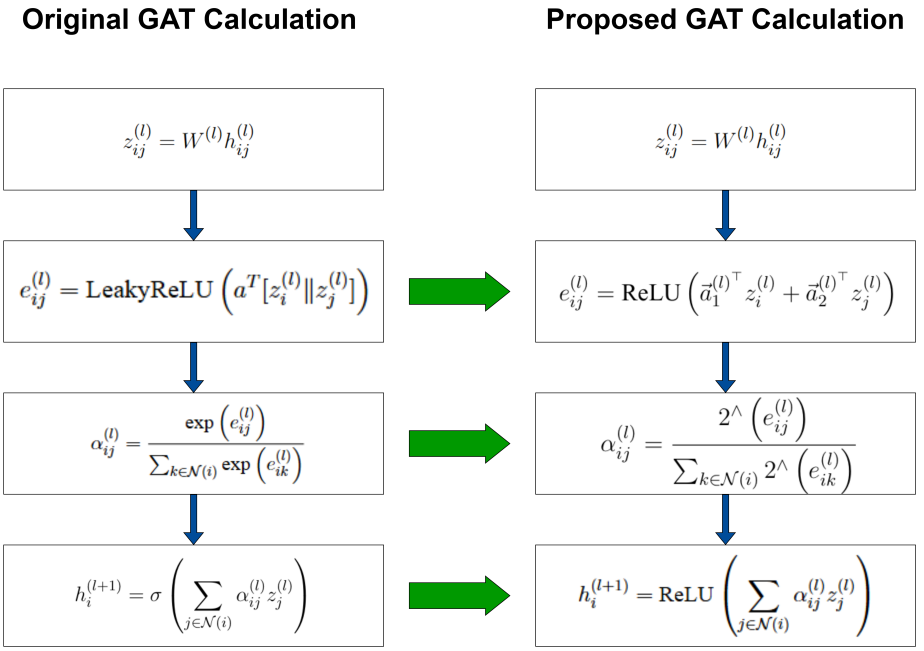


Figure 3.1: Proposed Modification in GAT Algorithm

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.

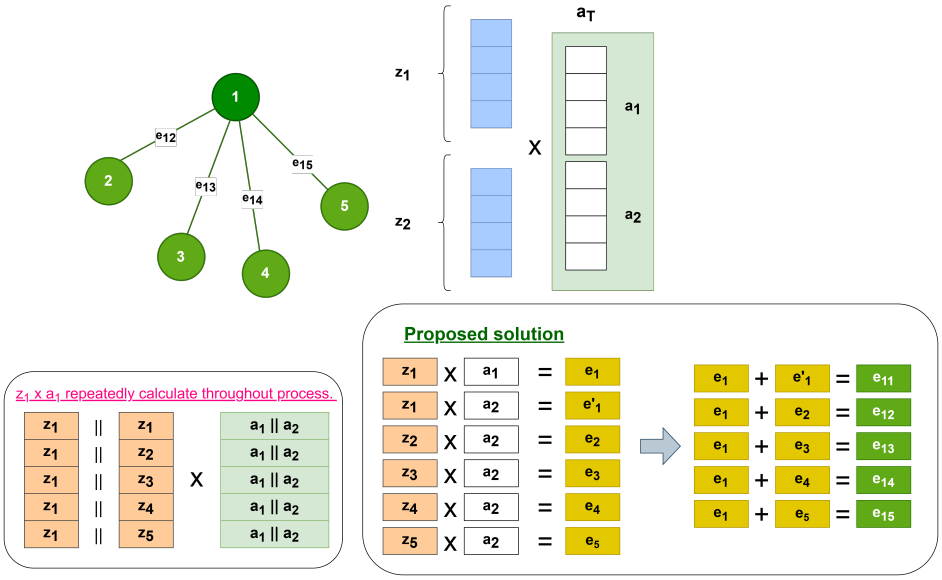


Figure 3.2: Issue when concatenating nodes and Proposed solution

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.

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.

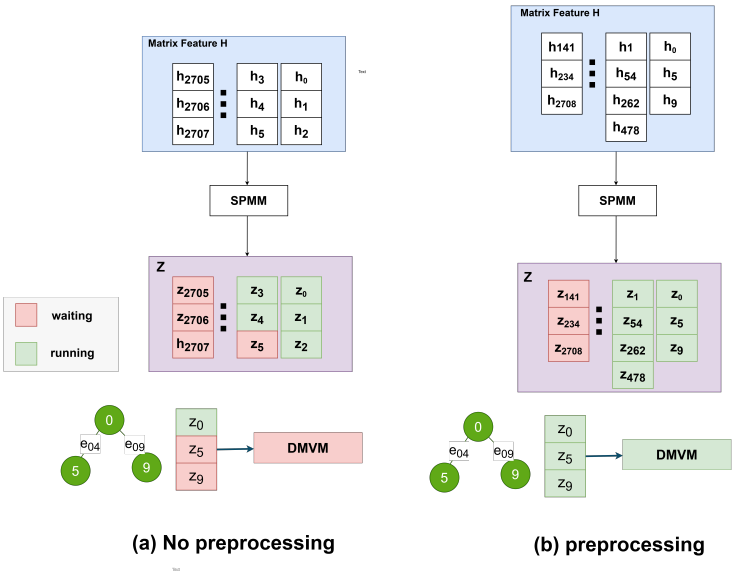


Figure 3.3: Illustration of no preprocessing and preprocessing data flow

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.

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.

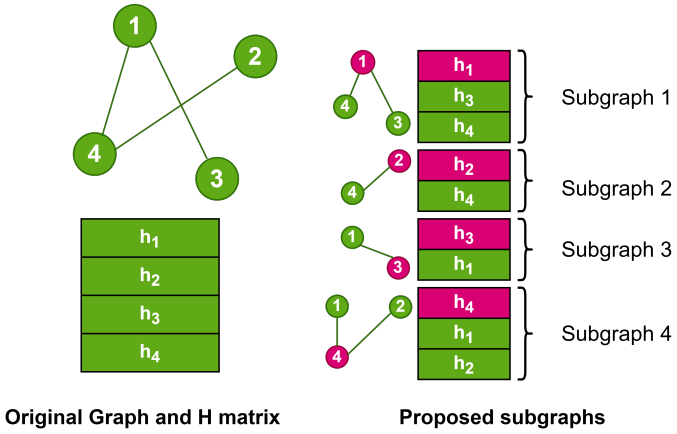


Figure 3.4: Demonstration of the extraction of a simple graph into subgraphs

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:

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

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

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%

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:

$$\mathbf{4 \| 3 \| 1}$$

equivalent value in binary format:

$$\mathbf{100 \| 011 \| 1}$$

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:

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

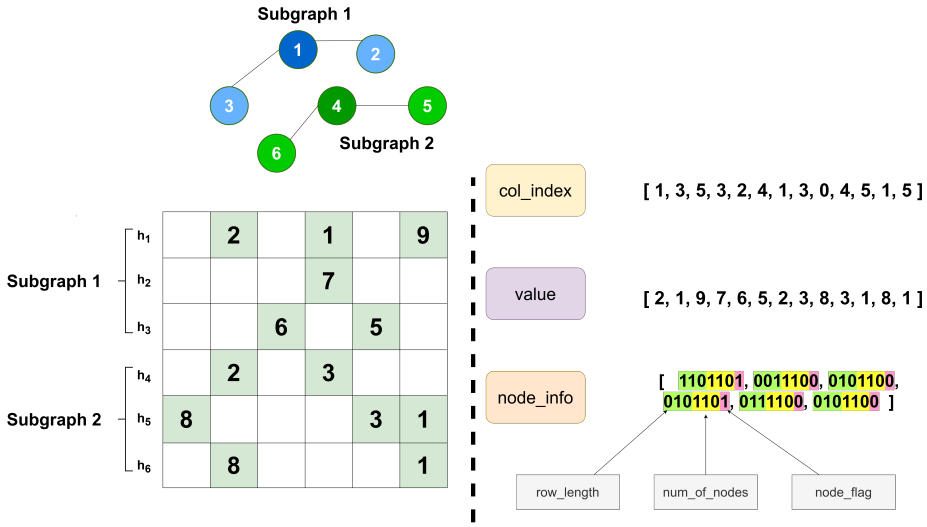


Figure 3.5: Proposed Modification in GAT Algorithm

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.

3.4. 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**.

Optimization Possibility

While FP32 provides high precision, this level of granularity is often excessive for the practical ranges of weights or activations encountered durx'ing 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, use lower-precision (e.g. **8-bit** instead of 32-bit float), leading to several benefits. More specifically, 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**.

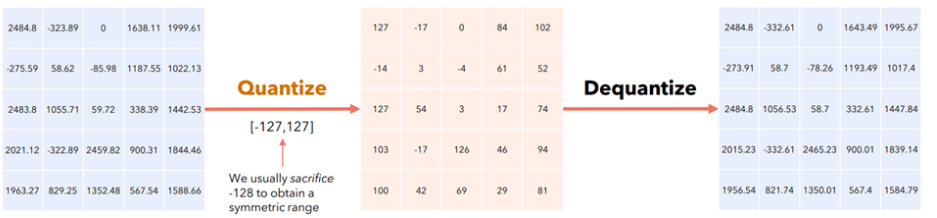


Figure 3.6: Quantization and Dequantization Process

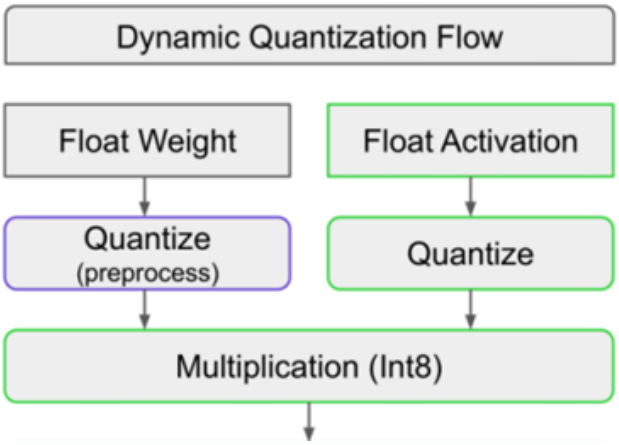


Figure 3.7: Dynamic Quantization Process

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 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**

What's different to dynamic quantization is that, **static quantization** determines the scales and zero points for all the activation tensors are **precomputed**. 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.

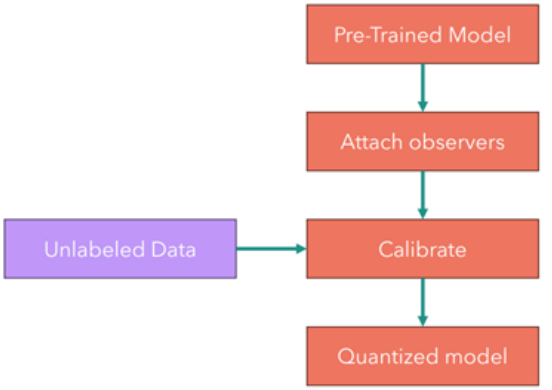


Figure 3.8: Static Quantization Process

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**

  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 infor-

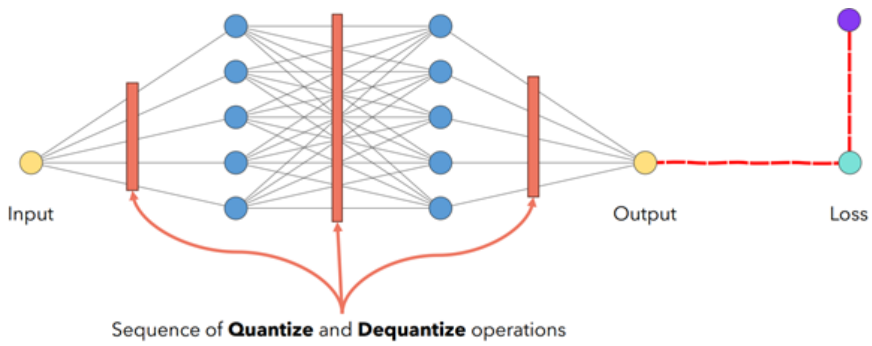


Figure 3.9: Quantization Aware Training Process

mation 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
- x is an unknown small value
- s is a scale point
- z is a zero point

If 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

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

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**
  - **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

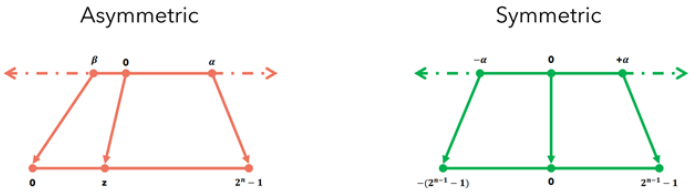


Figure 3.10: Range Mapping Symmetric and Assymetric

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.

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.

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.

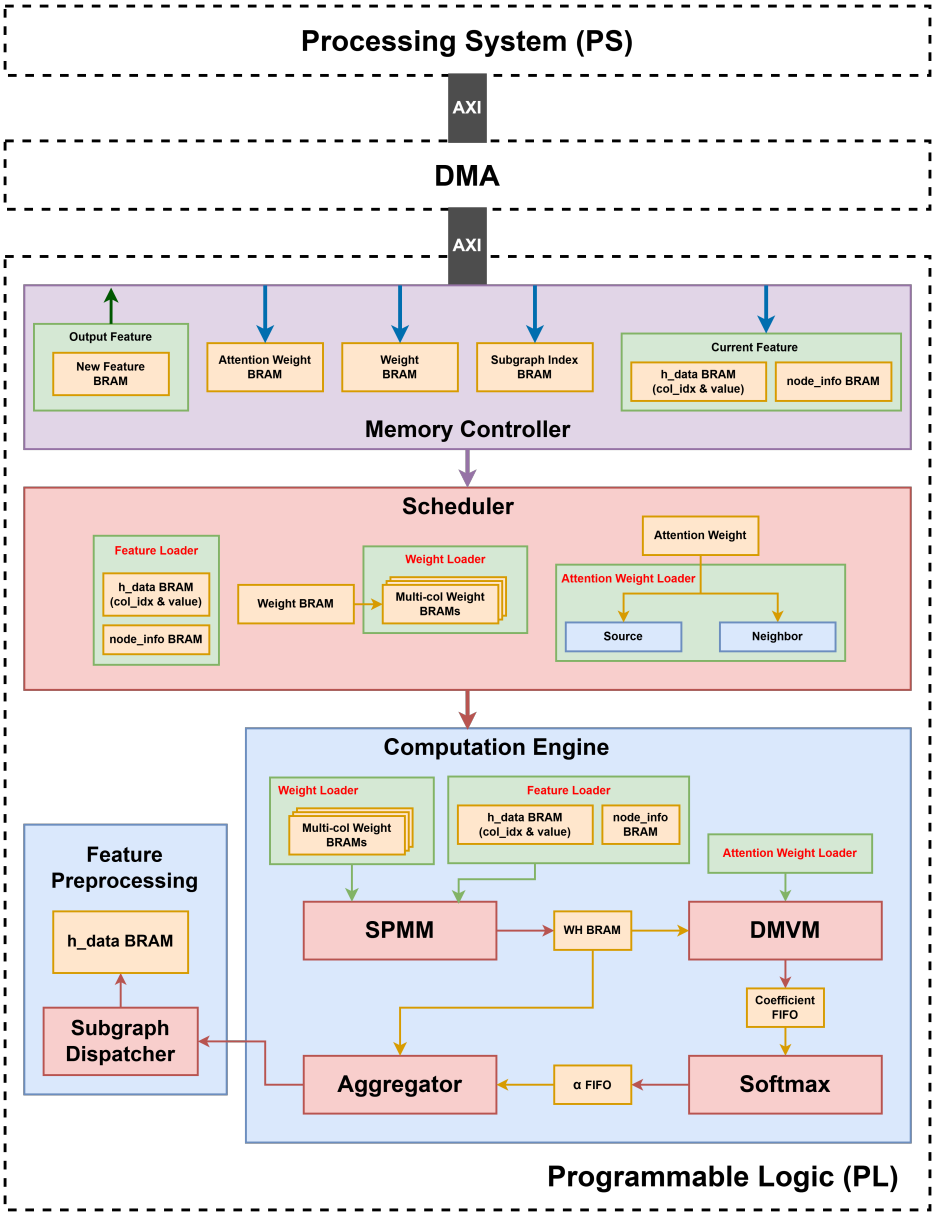


Figure 4.1: General Architecture of GAT accelerator

Figure 4.1 presents the overall architecture of the proposed GAT accelerator, which is divided into two main domains: the Processing System (PS) and the Programmable Logic (PL). The PS is responsible for data initialization, configuration, and high-level control, while the PL performs all core computations. Data is exchanged between the two domains through an AXI-based DMA interface.

The target model consists of two GAT layers, each with 16 hidden channels. The input to the system includes the feature matrix H, weight matrix W, attention vector a, and a subgraph index array used to define the computation topology for each batch. These inputs are loaded by the PS and transferred into PL-side BRAMs via DMA.

Within the PL, the Memory Controller organizes all data into dedicated BRAM blocks, including Subgraph Index BRAM, Feature BRAM, Weight BRAM, and Attention BRAM. The Scheduler manages data flow and controls access to these blocks during execution.

The compute pipeline comprises four stages: SPMM, DMVM, Softmax, and Aggregation. These modules are connected in a streaming manner to enable continuous processing of subgraph data. After the Aggregator finishes computing the updated features for Layer 1, the Subgraph Dispatcher reorganizes the results based on subgraph indices to match the input format required by Layer 2.

Once all computations are completed, the result features from the final layer are stored in BRAM. The PS can then read them back via AXI for final verification.

4.2. MEMORY CONTROLLER

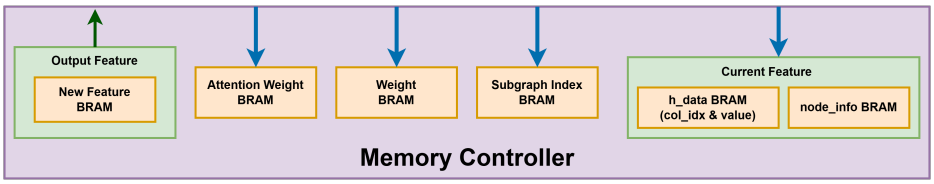


Figure 4.2: Memory Controller block diagram

Memory Controller serves as the central interface for storing and accessing all feature and parameter data between the PS and PL. It manages data movement across layers and supports efficient access during GAT computation by organizing intermediate and model data into multiple dedicated BRAM blocks.

At the beginning of execution, the PS sends four types of input data to the

PL via AXI-DMA and stores them into their corresponding BRAMs:

- **h_data BRAM**: Contains the input feature matrix H for the current layer, stored in the GCSR format. Each entry consists of a non-zero value and its associated column index (`col_idx` and `value`). After Layer 1 computation, this BRAM also receives preprocessed features from the Subgraph Dispatcher, which reorganizes the output from **New Feature BRAM** to match the subgraph structure required for Layer 2.
- **node_info BRAM**: Stores per-node metadata in the GCSR format, including `row_length` (number of non-zero elements), `num_node` (number of nodes in the current subgraph), and a `source_flag` indicating whether the node is a source or a neighbor.
- **Weight BRAM**: Stores both the weight matrices $W^{(1)}$, $W^{(2)}$ and the corresponding attention vectors $a^{(1)}$, $a^{(2)}$ for Layer 1 and Layer 2, sequentially arranged and transferred from the PS before execution begins.
- **subgraph_idx BRAM**: Contains the mapping from global nodes to subgraph locations, used during the reorganization phase by the Subgraph Dispatcher. Each entry includes an index value and a 1-bit `eog` (end-of-graph) flag to indicate the end of the mapping list for a given node.

During computation, these BRAMs are accessed in parallel by different compute modules. After a layer finishes execution, the updated feature vectors are first written to the **New Feature BRAM**. These results are then passed to the Subgraph Dispatcher, which reorganizes the features and writes them into **h_data BRAM** to serve as input for the next layer. Finally, the PS can read back the output features via AXI for final verification.

4.3. SCHEDULER

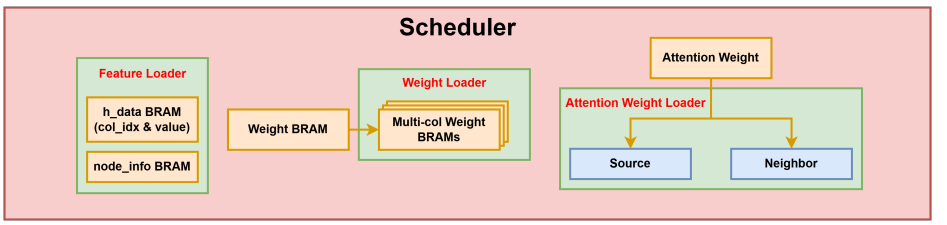


Figure 4.3: Scheduler block diagram

Scheduler is responsible for preprocessing and distributing data to appropriate computation modules after all inputs have been loaded into the BRAMs. It is structured into three main loaders: the **Feature Loader**, the **Weight Loader**, and the **Attention Weight Loader**. Each loader handles a specific type of data and ensures that the format and access mechanism are optimized for the downstream computation.

After the PS transfers the full Weight BRAM to PL, the Scheduler begins parsing and dispatching its contents based on their roles in the computation:

- For the **Weight Matrix** in **Layer 1**, the Scheduler allocates 16 separate BRAMs, each corresponding to one column of the matrix. This column-wise partitioning allows each SPMM processing element (SP-PE) to independently read its corresponding weight vector during the linear transformation stage $W \cdot H$.
- For the **Attention Weight vectors** $a^{(1)}$ and $a^{(2)}$, the Scheduler separates each vector into two register sets: one for source nodes and one for neighbor nodes. These values are broadcasted to the DMVM module for computing attention coefficients efficiently.
- For the **Weight Matrix** in **Layer 2**, the entire matrix is stored in a structured 3D array register, which supports fast retrieval during the second-layer Aggregator computation without requiring BRAM-based access.

In parallel, the Feature Loader begins reading from the **h_data BRAM**. After all weight columns for Layer 1 have been distributed, the computation for Layer 1 is triggered. Non-zero feature values are sequentially fetched on each clock cycle, with their corresponding column indices. At the same time, the **node_info BRAM** is accessed to retrieve row-level metadata, such as `row_length`, `num_node`, and `source_flag`, which are used to determine how many feature entries to read and how to route them within the computation pipeline.

4.4. COMPUTATION ENGINE

4.4.1. SPARSE MATRIX-VECTOR MULTIPLICATION

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:

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

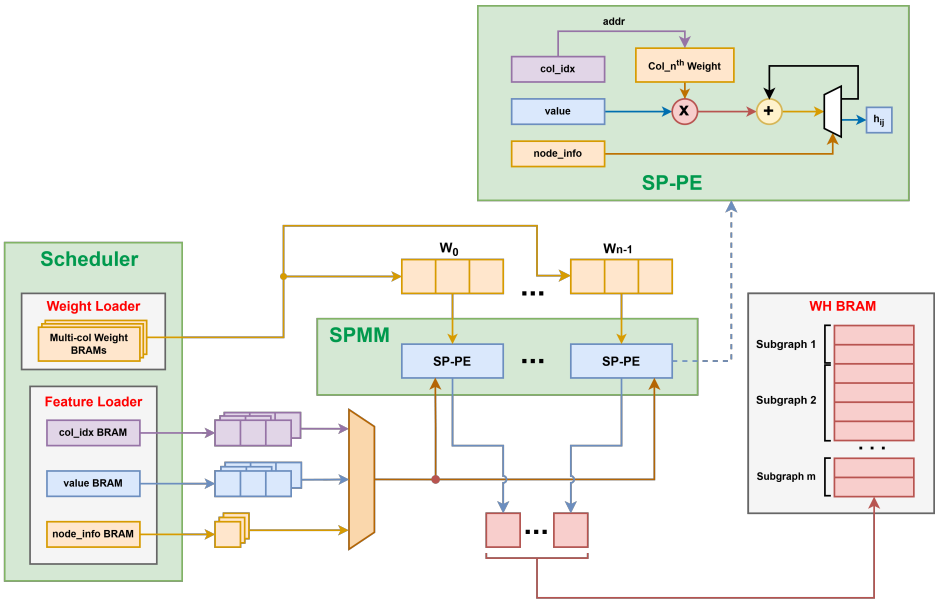


Figure 4.4: Hardware architecture of the SPMM module with 16 parallel SP-PEs and column-specific weight BRAMs

The module consists of the following components:

- **Feature Loader**: This module reads non-zero values and their column indices from the `h_data BRAM` and prepares them for processing by the **SP-PE** modules.
- **Weight Loader**: The *Weight Matrix* is stored in column-specific BRAMs. Each column is independently processed by the corresponding **SP-PE** module during the matrix-vector multiplication.
- **SP-PE** (Sparse Processing Element): Each of the 16 **SP-PE** modules performs the matrix-vector multiplication for one column of the weight matrix. The partial product is computed for each non-zero feature value and the corresponding weight, contributing to the final feature vector $W\vec{h}$.
- **WH BRAM**: The final result for the row is written into **WH BRAM**, where it is stored for further processing in the next computation stage.

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_nodes`), and whether the current node is a source node or a neighbor (`source_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_nodes` and `source_flag`, which will be used in the next computation steps.

4.4.2. DENSE MATRIX-VECTOR MULTIPLICATION

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:

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

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.

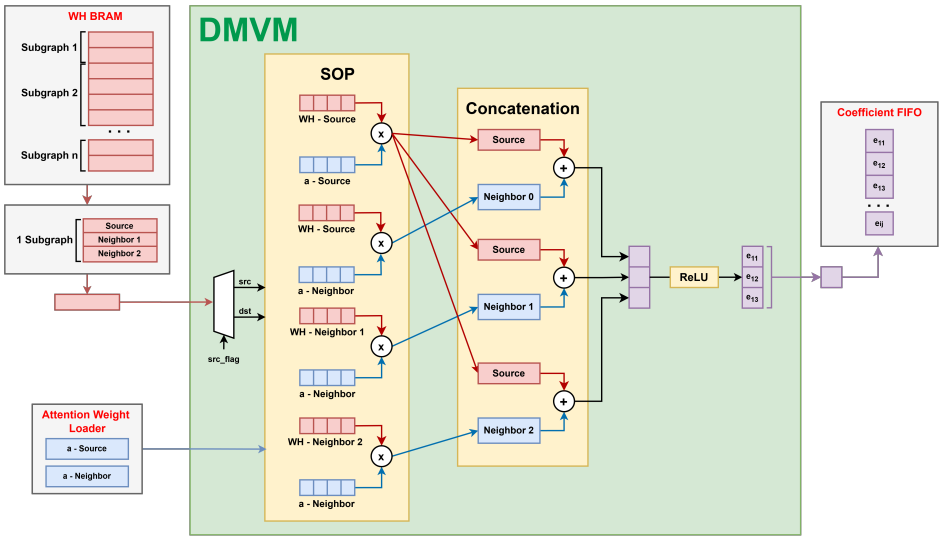


Figure 4.5: Hardware architecture of DMVM

In Figure 4.5, **DMVM** communicates with three main components:

- **WH BRAM**: Each entry contains a $W\vec{h}$ vector, along with `num_nodes` and `source_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 two separate vectors: one for source nodes and one for neighbor nodes.

- **Coefficient FIFO**: After calculating each attention coefficient e, the result is immediately pushed into the FIFO buffer for further processing without waiting for all coefficients of a subgraph to be computed.

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_flag`. Based on this flag, the $W\vec{h}$ vector is multiplied with either the source or neighbor `Attention Weight` vector.

- When a **source node** is fetched from the **WH BRAM**, it is multiplied with both the `a_source` and `a_neighbor` vectors. The result from the multiplication with `a_source` is immediately used to compute its own attention coefficient. Meanwhile, the result from the multiplication with `a_neighbor` is stored temporarily and reused to compute the attention coefficients of the following neighbor nodes within the same subgraph.
- For each **neighbor node**, its $W\vec{h}$ vector is multiplied with the corresponding `a_neighbor` vector. The result is then combined (added) with the stored result from the source node to complete the attention mechanism for the neighbor.
- **SOP** (Sum of Products) computes the dot product between vectors, consisting of:
  - **Product**: Each corresponding pair of elements is multiplied in a single clock cycle.
  - **Sum**: The results are accumulated using an Adder Tree structure, reducing the number of partial sums by half each cycle. Approximately $\log_2 N$ cycles are required to complete the sum.
- Each computed attention coefficient is passed through the **ReLU** activation function to eliminate negative values. The resulting coefficient is then immediately pushed into the **Coefficient FIFO**.
- Once the number of processed vectors reaches `num_nodes`, the system updates to the next subgraph, refreshing the stored source node results accordingly.

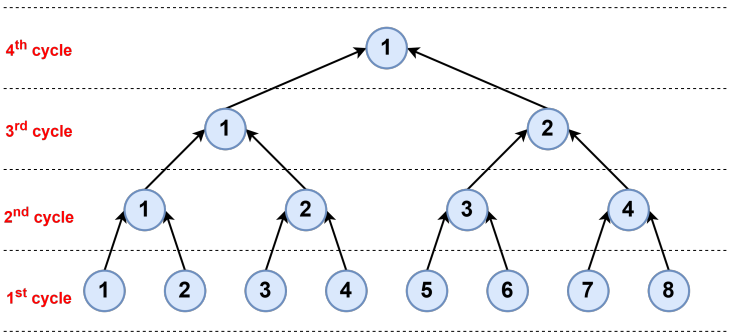


Figure 4.6: An example of sum calculation based on Adder Tree

4.4.3. SOFTMAX

Softmax is responsible for normalizing the attention coefficients e computed from the **DMVM** module, producing the final attention scores α_{ij} used in the aggregation step.

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

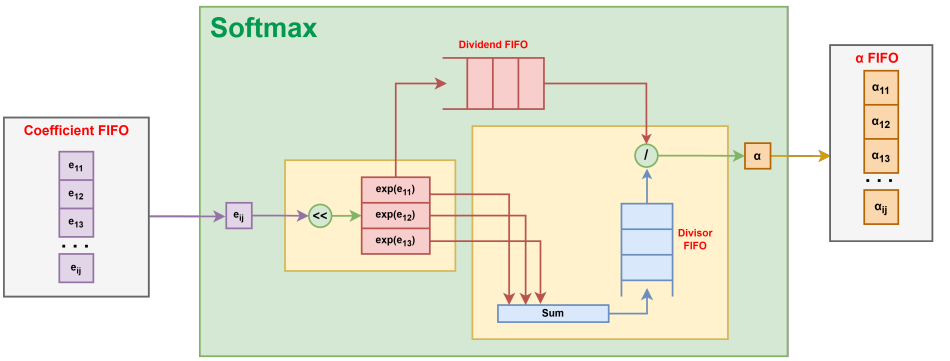


Figure 4.7: Hardware architecture of Softmax

In Figure 4.7, the **Softmax** module interfaces with four main FIFO buffers:

- **Coefficient FIFO**: Stores the attention coefficients e output from the **DMVM** module.
- **Dividend FIFO**: Stores each computed value of $2^{e_{ij}}$ corresponding to every node.
- **Divisor FIFO**: Stores the accumulated sum of $2^{e_{ij}}$ values within a subgraph.
- **Alpha FIFO**: Stores the normalized attention coefficients α_{ij} after they have been computed and normalized by the Softmax function.

To address the synchronization between coefficient generation and normalization, **Softmax** separates the write and read operations using independent FIFO buffers. This allows the two stages to operate asynchronously without introducing pipeline stalls or complex control dependencies.

- During the data collection phase, each attention coefficient e popped from the **Coefficient FIFO** is used to compute its base-2 exponential 2^e.

The result is pushed into the **Dividend FIFO**, while simultaneously being accumulated into a running sum, referred to as the Divisor. Once the number of processed nodes reaches `num_of_nodes` (corresponding to a complete subgraph), the Divisor is pushed into the **Divisor FIFO**.

- During the normalization phase, when the **Divisor FIFO** is not empty, one Divisor value is popped. Subsequently, the corresponding number of Dividend values are sequentially popped from the **Dividend FIFO**. For each Dividend, the normalized attention score α_{ij} is computed as:

$$\alpha_{ij} = \frac{2^{e_{ij}}}{\text{Divisor}}.$$

- The resulting α_{ij} values are pushed into the **Alpha FIFO** for subsequent aggregation.

4.4.4. 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 **FIFO**: Each entry in α **FIFO** 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.

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

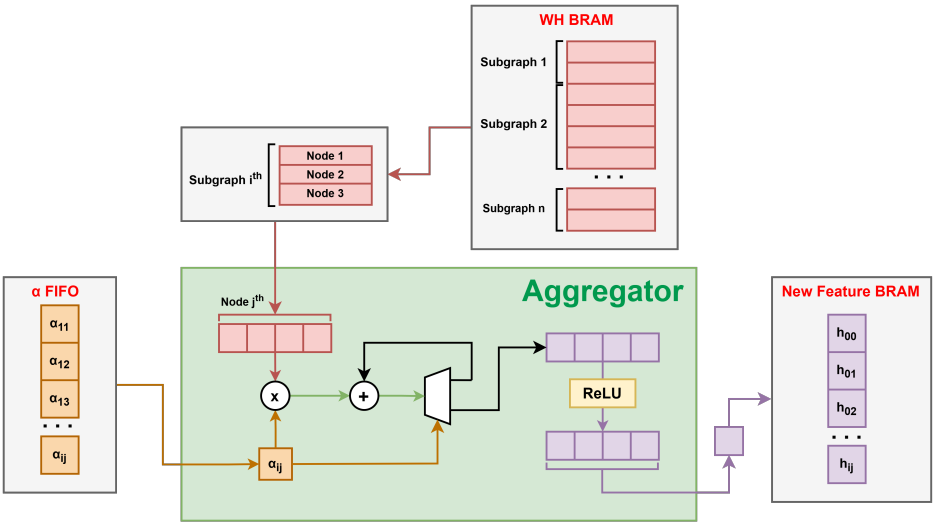


Figure 4.8: Hardware architecture of Aggregator.

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 **FIFO**. 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.

4.5. FEATURE PREPROCESSING

4.5.1. SUBGRAPH DISPATCHER

Subgraph Dispatcher is responsible for reorganizing the output feature vectors after Layer 1 computation, preparing the feature matrix format required for Layer 2 processing. This module ensures that each feature node is placed correctly based on the subgraph structure defined before computation.

As shown in Figure 4.9, the **Subgraph Dispatcher** interacts with three main memory blocks:

- **New Feature BRAM**: Stores the feature vectors (size 16) output from Layer 1 computation.
- **Subgraph Index BRAM**: Contains the list of target addresses where each feature node should be copied to. Each entry consists of an index value combined with an `eog` (end-of-graph) flag bit, indicating whether the current index is the last associated index for that feature node. Each index must be multiplied by 16 to calculate the correct base address in **h_data BRAM**.
- **h_data BRAM**: The destination memory where the reorganized feature vectors are stored for Layer 2 input.

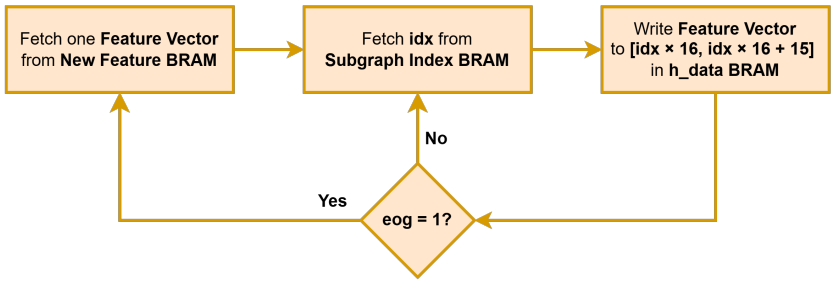


Figure 4.9: Subgraph Dispatcher Flow Chart

The operation of the **Subgraph Dispatcher** proceeds as follows:

- After all nodes complete their Layer 1 feature computation, the **Subgraph Dispatcher** begins operation. It sequentially reads feature vectors from the **New Feature BRAM**.
- For each feature node, it retrieves its corresponding list of subgraph indices from the **Subgraph Index BRAM**. Each index specifies a target location where the feature vector must be placed to fit the subgraph organization. The associated `eog` bit signals whether the current index is the last one for the current node.
- Each retrieved index is multiplied by 16 to calculate the actual base address in the **h_data BRAM**. The entire 16-element feature vector is then written into the calculated address range, spanning from `Address` to `Address+15`.

- If a feature node appears in multiple subgraphs, the dispatcher writes it to multiple addresses accordingly, iterating until an index with `eog = 1` is encountered. This ensures that each subgraph has a complete and independent copy of the necessary feature vectors.

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 5.5.

5.1. SYSTEM-ON-CHIP 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

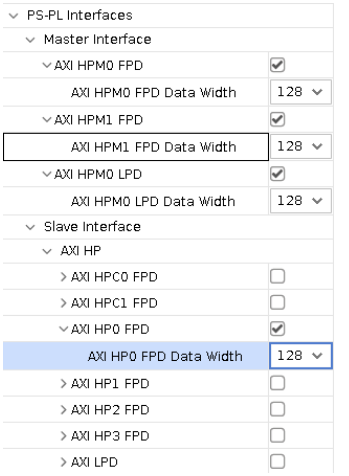


Figure 5.2: Interfaces configuration

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. DIRECT MEMORY ACCESS

In heterogeneous System-on-Chip (SoC) architectures like Xilinx's Zynq MPSoC, efficient communication between the Processing System (PS) and Programmable Logic (PL) is crucial for high-performance applications. To facilitate this, a combination of Direct Memory Access (DMA), Block RAM (BRAM), and AXI interconnects is employed to optimize data transfer and processing.

In systems utilizing Graph Attention Network (GAT) models, which are known for their computational complexity and large data volumes, efficient data handling mechanisms are essential. To address this, Block RAM (BRAM) within the Programmable Logic (PL) is employed for intermediate data storage and computation. Data transfer from the Processing System (PS) to the PL is managed by a Central Direct Memory Access (CDMA) controller, enabling high-speed, CPU-independent data movement.

Rather than consolidating all data into a single memory block, the archi-

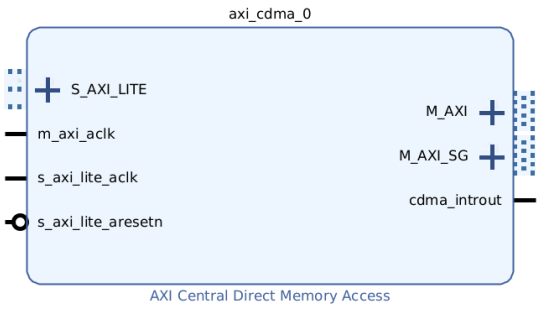


Figure 5.3: CDMA IP in Vivado

tecture distributes data across multiple BRAM controllers. This segmentation facilitates parallel processing and efficient resource utilization within the PL. Each BRAM controller is assigned a unique address space. To minimize overhead, contiguous address mapping should be, allowing the PS to initiate a single DMA transaction that the CDMA controller then distributes appropriately across the BRAMs.

However, empirical observations indicate that, in this specific system configuration, the performance difference between transferring data to contiguous versus non-contiguous memory addresses is negligible. This suggests that the DMA engine and memory architecture are effectively optimized to handle both types of transfers with similar efficiency, providing flexibility in memory allocation strategies without compromising performance.

Nonetheless, when implementing multiple BRAM blocks with varying size configurations, especially under non-contiguous addressing, one implementation level limitation must be considered. Specifically, Vivado enforces alignment constraints on the base addresses of each memory-mapped AXI BRAM Controller. The base address must be a multiple of the assigned address range (i.e., aligned with the range size).

For example, if the first BRAM is configured at address `0xC000_0000` with a range of 1MB, and a second BRAM is placed at `0xC001_0000` with a smaller range of 256KB, Vivado rejects this configuration. The reason is that `0xC001_0000` is not a multiple of 256KB (`0x40000`), thereby violating the required address alignment.

To resolve this, each BRAM's base address must be carefully calculated to satisfy both non-overlapping constraints and alignment requirements. Fail-

ure to do so may result in address decoding conflicts or unrecognized memory blocks during synthesis and implementation.

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.

5.1.3. REGISTER BANK CORE

During the implementation of the acceleration core, direct interaction between the Zynq PS and the core's internal signals or values is not possible, as the PS lacks a dedicated datapath for such access. Therefore, to enable configuration and control from the software side, a custom IP module containing memory-mapped registers is implemented. This IP allows the Zynq PS to read-/write registers via the MMIO (Memory-Mapped I/O) library, forming the communication bridge between the software and hardware domains.

The Register Bank consists of 32-bit registers and supports access from both Processing System (PS) and Programmable Logic (PL). The PS interfaces with the Register Bank through an AXI memory-mapped bus, while the PL accesses the registers directly using logic-level signals.

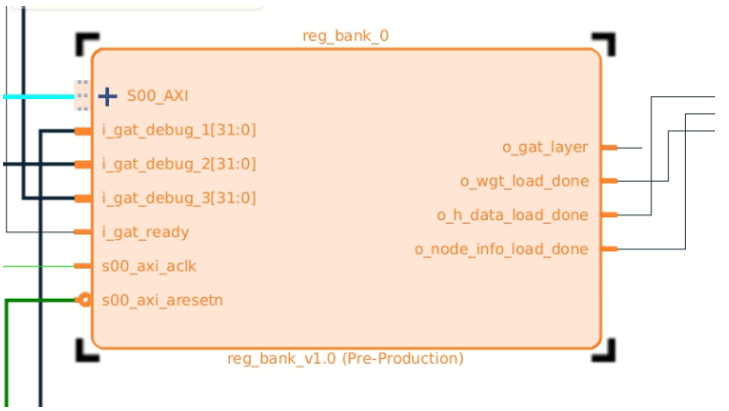


Figure 5.4: Register Bank

The key interfaces are as follows:

1. **S00_AXI**

This is the main interface that connects the Register Bank to the Zynq PS. It complies with the AXI4-Lite protocol and allows software to perform memory-mapped read and write operations to configure and control the hardware core. The PS communicates with this interface using the MMIO library at the software level.

2. **i_gat_ready**

   This input signal originates from the acceleration core and is used to indicate whether the core is finish its calculation and be ready for CDMA to transfer back to Zynq PS Memory.

3. **o_gat_layer**

   This output signal is generated based on a software configuration value written by the PS. It is typically used to specify the target layer of computation to the hardware logic.

   In this research, two computation methods are implemented. In **Method 2**, where two layers are processed after a single data transfer phase, this signal is **not used** and can be ignored.

4. **o_load_done**

   This signal is asserted when the Register Bank confirms that all configuration and data loading steps (e.g., weight loading, node info setup) have been completed. It is used by the acceleration core as a trigger to begin execution.

5.1.4. OVERALL BLOCK DESIGN

The overall system is constructed by integrating multiple IP blocks within the Vivado block design environment. These components include the Zynq Processing System (PS), AXI Interconnect, Central Direct Memory Access (CDMA), AXI SmartConnect, AXI BRAM Controllers, Block RAMs (BRAMs), the custom Register Bank, and the top-level acceleration core. The Zynq PS serves as the control unit and initiator of all transactions. It interfaces with both the CDMA and the Register Bank through AXI master ports. Specifically:

- **M_AXI_HPM0_FPD** is connected to the AXI Interconnect, which routes transactions to the CDMA and Register Bank.
- **CDMA** acts as the DMA engine, performing high-speed memory transfers from DRAM to BRAM without CPU involvement.

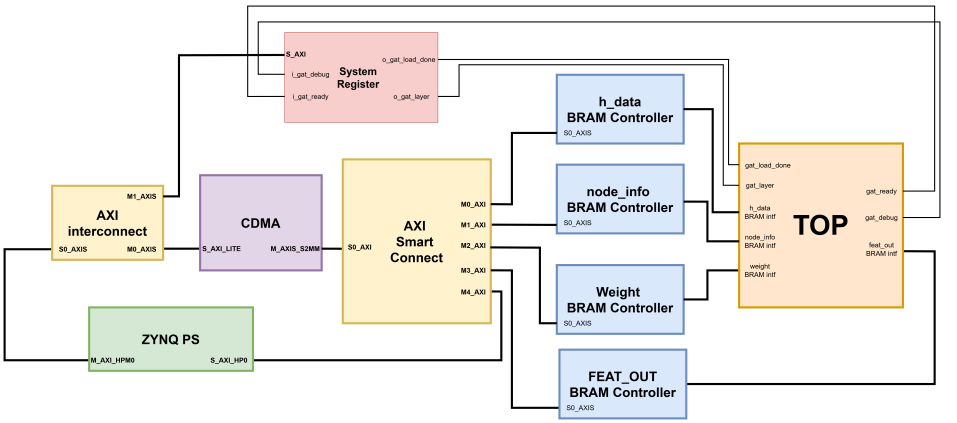


Figure 5.5: Overall block design

5

- The **AXI Interconnect** provides connectivity between the Zynq PS and multiple slave peripherals, including the Register Bank and the CDMA engine.

On the data sink side, the CDMA is connected to multiple **AXI BRAM Controllers** through an **AXI SmartConnect**. Each controller manages access to a dedicated BRAM block. This segmented memory design enables parallel read-/write operations and better utilization of PL resources.

The **Register Bank** is mapped into the PS address space and accessed via MMIO. It provides configuration, control, and status monitoring functionalities for the acceleration core. Signals such as `o_gat_layer`, `o_load_done`, and `i_gat_ready` are connected directly between the Register Bank and the top-level core.

All modules are synchronized through a common clock domain sourced from the `pl_clk0` output of the Zynq PS. Reset signals are propagated using the Processor System Reset IP, ensuring deterministic system startup.

At the top level, the acceleration core receives its configuration and control signals from the Register Bank and fetches input data from the BRAMs populated by CDMA. The results can then be transferred back to DRAM via a reverse DMA transaction, completing the computation cycle.

Table 5.1: Address and Memory Map

Register Name	Interface	Slave Segment	Base Address	Range	High Address
Sysreg	S00_AXI	S00_AXI_reg	0xA001_0000	64K	0xA001_FFFF
AXI CDMA S_AXI_LITE	S_AXI_LITE	Reg	0xA000_0000	64K	0xA000_FFFF
Output Data (feat_out)	S_AXI	Mem0	0xE800_0000	256K	0xE803_FFFF
h_data	S_AXI	Mem0	0xE000_0000	2M	0xE01F_FFFF
node_info	S_AXI	Mem0	0xE200_0000	64K	0xE200_FFFF
subgraph_idx	S_AXI	Mem0	0xE600_0000	64K	0xE600_FFFF
wgt_a	S_AXI	Mem0	0xE400_0000	256K	0xE403_FFFF
DDR Low	S_AXI_HP0_FPD	HP0_DDR_LOW	0x0000_0000	2G	0x7FFF_FFFF
QSPI	S_AXI_HP0_FPD	HP0_QSPI	0xC100_0000	16M	0xC1FF_FFFF

5.2. PYNQ FRAMEWORK IMPLEMENTATION

5.2.1. PYNQ OVERLAY

An Overlay is a pre-configured FPGA hardware design (bitstream) that defines the programmable logic (PL) architecture. It includes intellectual property (IP) cores, custom accelerators, and peripheral interfaces, which can be dynamically loaded onto the FPGA without rebooting the system. Overlays abstract low-level hardware details, allowing developers to interact with FPGA resources via Python APIs. An Overlay is instantiated by providing the path to a `.bit` file (and optionally a `.hwh` file) using the **`Overlay(bitstream_path)`** class, automatically mapping hardware blocks to accessible Python objects.

Usage:

```
from pynq import Overlay
ol = Overlay("your_design.bit")
```

5.2.2. PYNQ MMIO

Memory-Mapped I/O (MMIO) maps an AXI-Lite address range into user space via the **`pynq.MMIO`** class, permitting low-latency **`read(offset)`** and **`write(offset, value)`** transactions to peripheral registers directly from Python. It requires specifying the base **`address`** and the **`size`** of the memory region during initialization. MMIO provides a direct interface for fine-grained control of custom IP cores deployed on the PL.

Usage:

```
from pynq import MMIO
mmio = MMIO(address=0x40000000, size=0x1000)
mmio.write(0x0, 0xDEADBEEF)
value = mmio.read(0x0)
```

5.2.3. PYNQ ALLOCATION

The `pynq.allocate` helper reserves a physically-contiguous buffer in PS DDR, returns a NumPy-compatible object whose `device_address` field can be passed to PL masters, and provides `flush()` and `invalidate()` methods for cache coherence management. It accepts `shape` and `dtype` parameters to define the size and type of the allocated buffer, enabling efficient interaction between processing systems and hardware accelerators.

Usage:

```
from pynq import allocate
import numpy as np

buf = allocate(shape=(1024,), dtype=np.float32)
buf[:] = np.ones(1024, dtype=np.float32)
buf.flush()
```

5

5.2.4. PYNQ CDMA

The community package `pynq_cdma` exposes the AXI Central Direct Memory Access (CDMA) IP via a simple `transfer(src_addr, dst_addr, length)` routine, enabling high-bandwidth memory-to-memory copies inside the PL. Users instantiate the `CDMA(base_address)` class by providing the base AXI address of the CDMA IP core. The `idle` property reports whether the transfer is complete, allowing non-blocking status checking.

Usage:

```
from pynq_cdma import CDMA
cdma = CDMA(base_address=0xA0000000)

cdma.transfer(src_buf.device_address, dst_buf.device_address,
    length=4096)
while not cdma.idle:
    pass
```

Figure 5.6 illustrates the overall flow of the data transferring process using PYNQ libraries and the AXI CDMA engine. Initially, input buffers are allocated in the Processing System (PS) memory through the pynq.allocate() method. Control registers are configured via pynq.MMIO.write() to coordinate the loading operation. Subsequently, critical data structures, including model weights, feature matrices, and graph information, are transferred into the Programmable Logic (PL) using pynq_cdma.CDMA.transfer(). Upon successful data transfer, a synchronization signal is issued to trigger computation in the accelerator. The system continuously polls the accelerator status using pynq.MMIO.read() until execution is completed, after which the output results are retrieved.

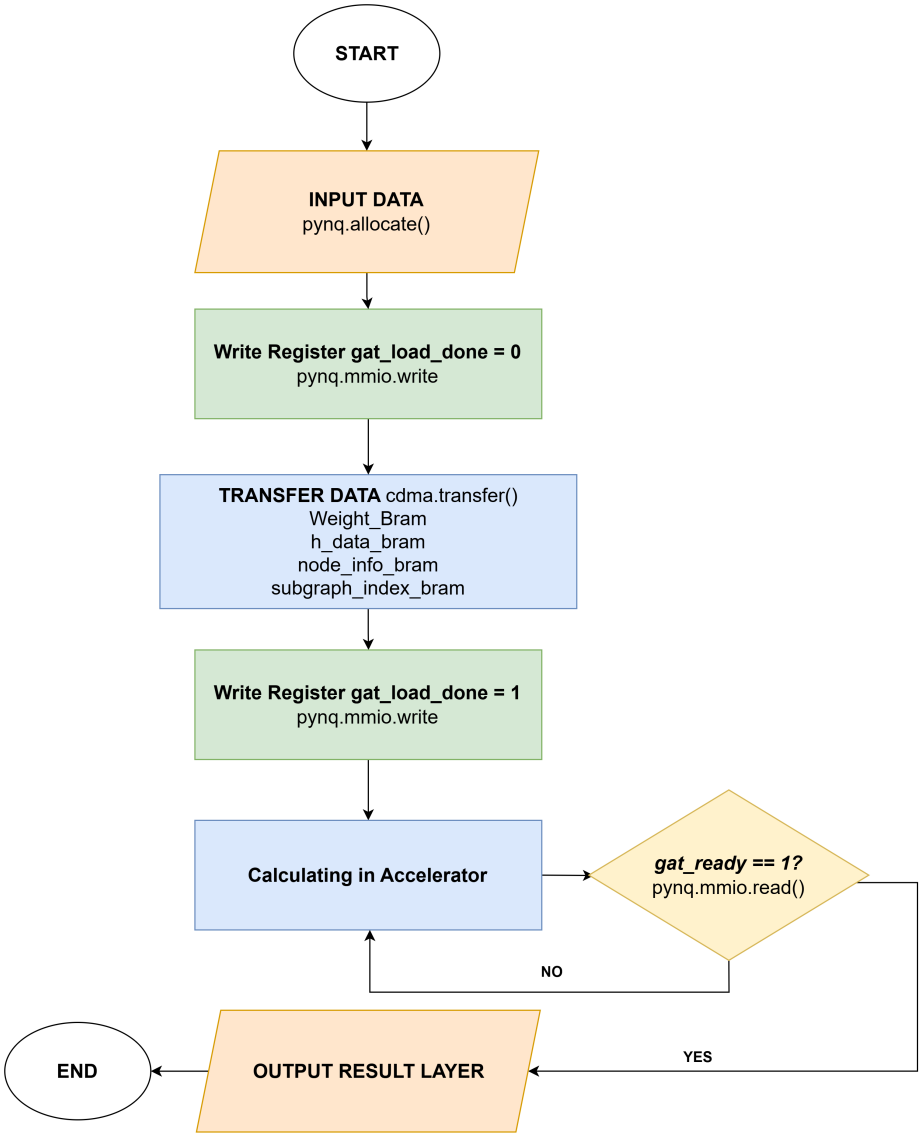


Figure 5.6: Transferring Flow using PYNQ CDMA and PYNQ Libraries

All the algorithms of the transferring flow are integrated into a Python Class called Accelerator with some methods supported: `prepare_data()` is for preparing input data from text files and validate it; `transfer()` is for transferring and record transferring time and execution time of the accelerator.

```
class Accelerator:
def transfer(self, layer):
  if layer == 2:
    self.sysreg.write(REG["gat_load_done"], 0)
    trans_start_time = time.perf_counter()
    self.cdma.transfer(self.node_info_bram.buffer, self.
        node_info_bram.BASE_ADDR)
    self.cdma.transfer(self.h_data_bram.buffer, self.
        h_data_bram.BASE_ADDR)
    self.cdma.transfer(self.weight_bram.buffer, self.
        weight_bram.BASE_ADDR)
    self.cdma.transfer(self.subgraph_index_bram.buffer, self.
        subgraph_index_bram.BASE_ADDR)
    trans_end_time = time.perf_counter()
    self.sysreg.write(REG["gat_load_done"], 1)
    print("\n[Accelerator] : ",f"Transfering Time = {round((
        trans_end_time-trans_start_time)*1000, 3)} ms")
    #=================================
    start_time = time.perf_counter()
    while (1):
      if self.sysreg.read(REG["gat_ready"]) == 1:
        end_time = time.perf_counter()
        break

    self.cdma.transfer(self.feat_out_bram.BASE_ADDR, self.
        feat_out_bram.buffer)
    self.sysreg.write(REG["gat_load_done"], 0)
    print("[Accelerator] : ",f"Execution Time = {round((
        end_time-start_time)*1000, 3)} ms\n")
    #=================================
    self.result_final_layer = []
    for i in range(len(self.feat_out_bram.buffer)):
      self.result_final_layer.append(self.feat_out_bram.buffer
          [i] / (2**16))
    print("DONE")
    return
```

Listing 5.1: Implementation of a transferring flow

Implementation

```
In [13]: accelerator = Accelerator(overlay_gat, sysreg_ip, BramLayer1)

In [14]: accelerator.prepare_data(full_layer)
         accelerator.transfer(full_layer)

         [LoadData] : reading /root/GAT_FPGA/gat_v2/data/citeseer//layer_1/input/h_data.txt
         [LoadData] : reading /root/GAT_FPGA/gat_v2/data/citeseer//layer_1/input/node_info.txt
         [LoadData] : reading /root/GAT_FPGA/gat_v2/data/citeseer//layer_1/input/weight.txt
         [LoadData] : reading /root/GAT_FPGA/gat_v2/data/citeseer//layer_1/input/subgraph_index.txt

         [Accelerator] :  Transfering Time = 11.608 ms
         [Accelerator] :  Execution Time = 3.617 ms
```

Figure 5.7: Example of a Accelerator Flow

5.3. SOFTWARE IMPLEMENTATION

5.3.1. GAT TRAINING

DATASET

Cora Dataset

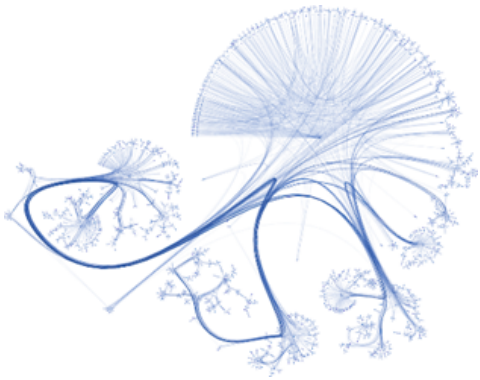


Figure 5.8: A visualization of the Cora citation network generated using yEd Live. Visualization adapted from https://graphsandnetworks.com/the-cora-dataset

- **Type**: Citation Network.
- **Description**: The Cora data set is one of the most popular data sets in the citation network to evaluate 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 data set 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) **(2,008 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).
- **Isolated Nodes**: None

CiteSeer Dataset

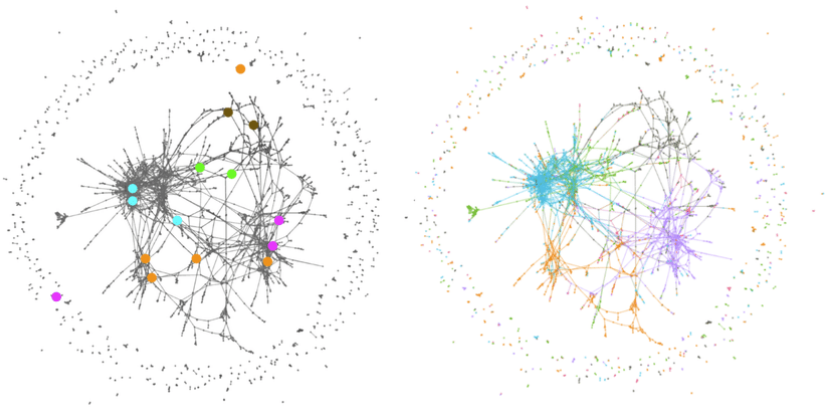


Figure 5.9: Visualization of the CiteSeer citation network showing document nodes and citation edges. Visualization adapted from ResearchGate.

- **Type**: Citation Network.
- **Description**: The CiteSeer data set is a widely used benchmark in graph-based machine learning tasks. It contains scientific publications (nodes) from a digital library of computer and information science literature, connected through citation links (edges). Each document is represented as a node with a bag-of-words feature vector describing its textual content. CiteSeer is commonly used for node classification tasks, where the aim is to predict the subject category of each paper based on its content and its citation relationships. The nodes are categorized into one of 6 different research topics such as AI, databases, information retrieval, etc.

- **Nodes**: Represent documents (research papers) **(3,327 nodes)**.
- **Edges**: Represent citations between papers **(4,732 edges)**.
- **Features**: Each node has a bag-of-words feature vector **(3,703-dimensional binary vector)..**
- **Classes**: Correspond to different computer science topics (e.g., AI, database systems, information retrieval, etc.) **(6 classes)**.
- **Task**: Node classification (assign each node a label from one of the 6 classes).
- **Isolated Nodes**: CiteSeer contains a number of isolated nodes — nodes that have no incoming or outgoing edges (no citations to or from other papers).

Dataset Selection Explanation

In this project, two datasets — **Cora** and **CiteSeer** — are used to comprehensively evaluate the performance and robustness of the graph model.

- **Cora** serves as a relatively **simple** and **clean** dataset. It features a fully connected citation network without isolated nodes, making it ideal for initial model development, debugging, and verifying basic functionality in controlled conditions.
- **CiteSeer**, on the other hand, represents a **more advanced** and **realistic** scenario. It contains isolated nodes (nodes without any connections), reflecting the kinds of irregularities often encountered in real-world graph data, such as missing links, incomplete information, or sparsely connected networks.

TRAINING MODELS

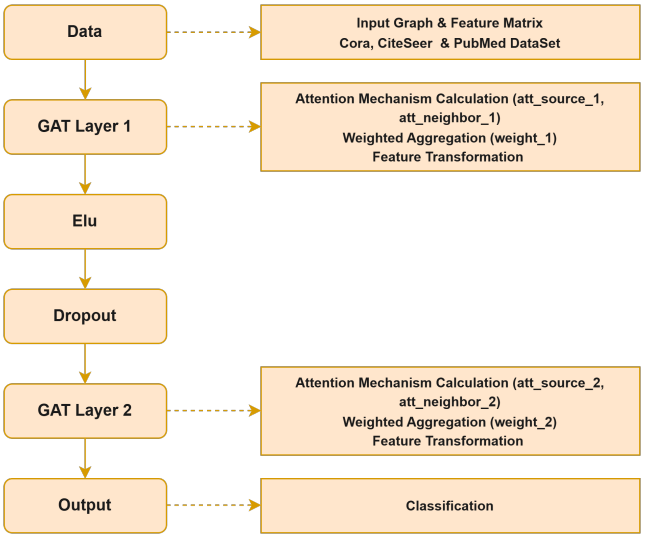


Figure 5.10: Two Layer GAT Training Model

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

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 gradi-

ents, 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.

```
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
```

Listing 5.2: Implementation through GAT class

5.3.2. QUANTIZATION

```
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)
```

Listing 5.3: Implementation through BuildModel class

During this phrase, we apply **self-aware quantization (QAT)** to optimize the GAT model by reducing 32-bit floating-point computations to 8-bit inte-

gers. 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.

5.3.3. 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.

```
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': ...
        }
        ...
```

Listing 5.4: Implementation through GCSR_Data_Compression_Builder class

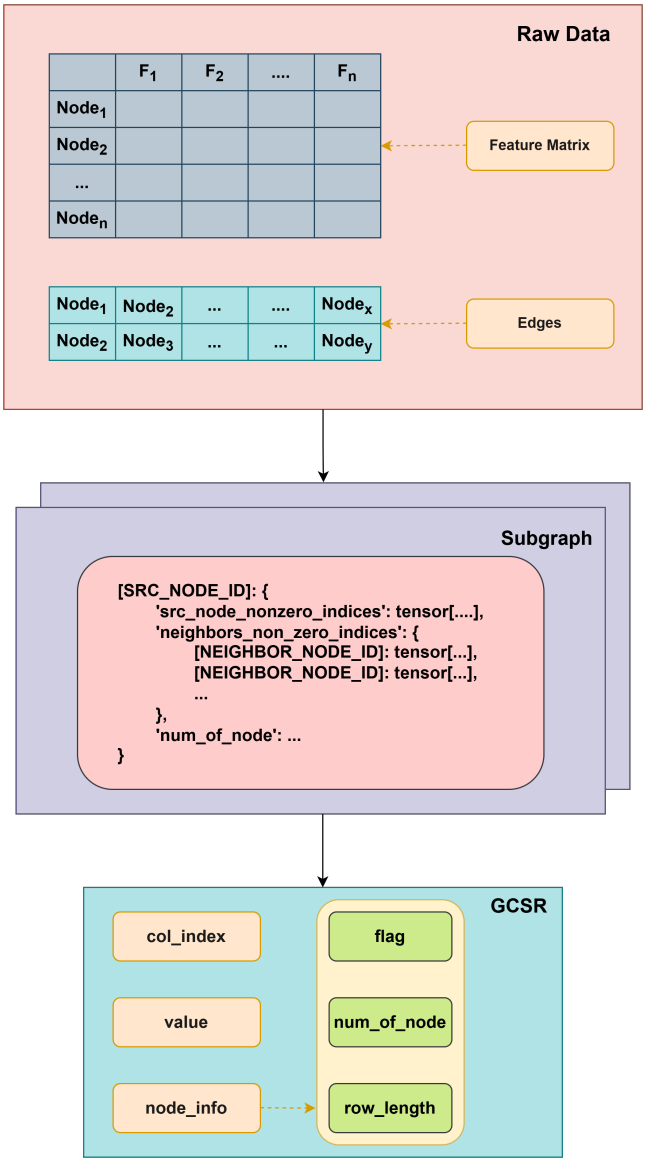


Figure 5.11: GCSR – Data Compression Builder

6

EXPERIMENT RESULTS

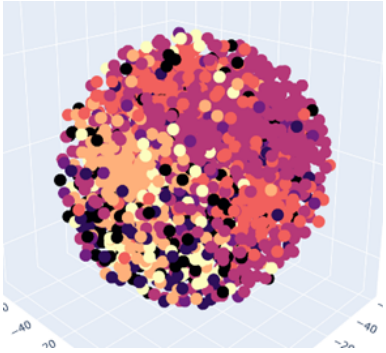
6.1. MODEL TRAINING

6.1.1. WITHOUT QUANTIZATION

VISUALIZE DATASET WITHOUT TRAINING MODEL

Figures 6.1 and 6.2 display **2D** and **3D** projections of the **raw Cora** and **CiteSeer** datasets, respectively, prior to any preprocessing or model training. These visualizations highlight the absence of distinct class separability, structural patterns, or discernible clusters within the untransformed data, emphasizing the challenge of direct classification without further feature engineering.

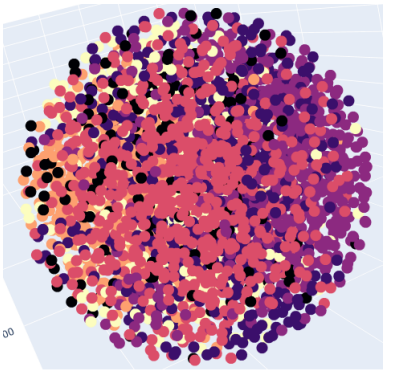
(a) Cora Dataset: 2D Visualization (Untrained Model)



(b) Cora Dataset: 3D Visualization (Untrained Model)

Figure 6.1: Visualization of the **Cora** Dataset in its raw form, prior to any model training or feature transformations.

(a) CiteSeer Dataset: 2D Visualization (Untrained Model)



(b) CiteSeer Dataset: 3D Visualization (Untrained Model)

Figure 6.2: Visualization of the **CiteSeer** Dataset in its raw form, prior to any model training or feature transformations.

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:

The figure 6.3 below presents the training performance of the GAT model on the **Cora** dataset, achieving an accuracy of approximately **0.807**. This result was obtained under full-precision conditions, **without the application of any quantization techniques**, serving as a baseline for evaluating the impact of quantization methods such as Quantization Aware Training (QAT) on model accuracy and representational capacity.

```
[41] test_accuracy = buildGATModel.test(visualization_2D=False, visualization_3D=False)
     print(f'Test Accuracy without Quantization Aware Training (QAT): {test_accuracy}')

Test Accuracy without Quantization Aware Training (QAT): 0.807
```

Figure 6.3: Accuracy after training model on the **Cora** Dataset - **without Quantization Aware Training**

The figure below demonstrates the training accuracy of the GAT model on the **CiteSeer** dataset, reaching approximately **0.705**. This accuracy was achieved under standard full-precision training conditions, **with no quantization techniques applied**, and serves as a reference point for assessing the impact of quantization-aware methods on model performance.

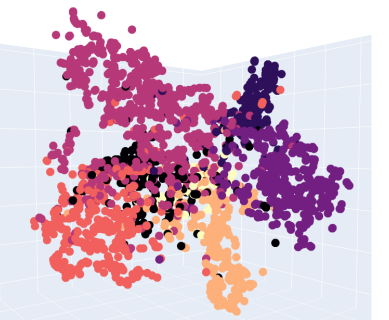

```
[45] test_accuracy = buildGATModel.test(visualization_2D=False, visualization_3D=False)
     print(f'Test Accuracy without Quantization Aware Training (QAT): {test_accuracy}')

Test Accuracy without Quantization Aware Training (QAT): 0.705
```

Figure 6.4: Accuracy after training model on the **CiteSeer** Dataset - **without Quantization Aware Training**

VISUALIZATION RESULT CLASSIFICATION

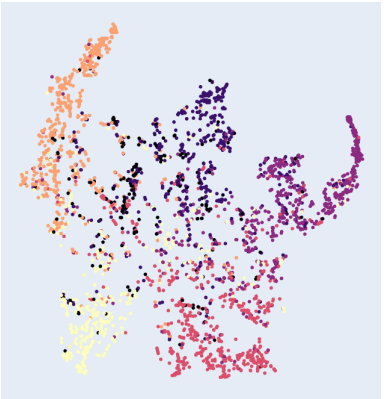
Figures 6.5 and 6.6 visualize the classification results of the Cora and CiteSeer datasets, respectively, achieved with an accuracy of **0.807** and **0.705** in both 2D and 3D projections. These results are obtained **without applying Quantization Aware Training (QAT)**. They serve as **baseline performance metrics** for comparison against subsequent experiments where quantization techniques are applied, allowing for an evaluation of their **impact on classification accuracy**.

(a) **Cora** dataset: 2D Visualization **(Without Quantization)**

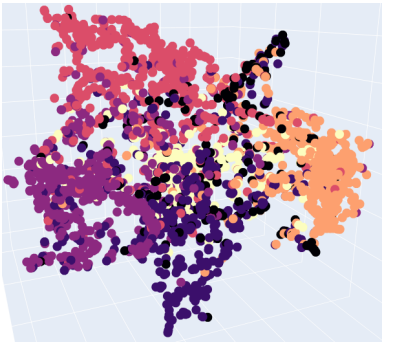


(b) **Cora** dataset: 3D Visualization (Without Quantization)

Figure 6.5: Visualization of classification results on the **Cora** dataset **(Without Quantization)**



(a) **CiteSeer** dataset: 2D Visualization **(Without Quantization)**



(b) **CiteSeer** dataset: 3D Visualization **(Without Quantization)**

Figure 6.6: Visualization of classification results on the **CiteSeer** dataset **(Without Quantization)**

6.1.2. WITH QUANTIZATION

TRAINING GAT MODEL

Additional Step:

However, the current training and inference processes are conducted using 32-bit floating-point (FP32) precision, which, while accurate, imposes significant computational and memory overhead. To address this, **Fake Quantization** layers will be integrated into the model. These layers simulate low-precision

inference by sequentially applying **quantization and dequantization** operations during training. This approach enables the model to adapt to reduced-precision constraints by emulating **8-bit integer (INT8)** arithmetic, forming the core of QAT methodology. QAT ensures that the model learns to compensate for quantization effects during training, thereby preserving performance while significantly **reducing computational cost and memory usage**.

Outcome:

The figure 6.7 below presents the training results of the GAT model on the **Cora** dataset **with Quantization Aware Training (QAT)** applied. Despite the introduction of quantization constraints during training, the model **maintains a high level of performance**, achieving an accuracy of approximately **0.805**, which is **only marginally lower than** the baseline accuracy of **0.807** obtained without quantization. This **minimal drop** demonstrates the robustness of the GAT model to quantization and highlights the effectiveness of QAT in **preserving model accuracy under reduced-precision conditions**:

```
[50] test_accuracy = buildGATModel.test(visualization_2D=False, visualization_3D=False)
     print(f'Test Accuracy with Quantization Aware Training (QAT): {test_accuracy}')

Test Accuracy with Quantization Aware Training (QAT): 0.805
```

Figure 6.7: Accuracy after training model - with **Quantization Aware Training** on **Cora** dataset

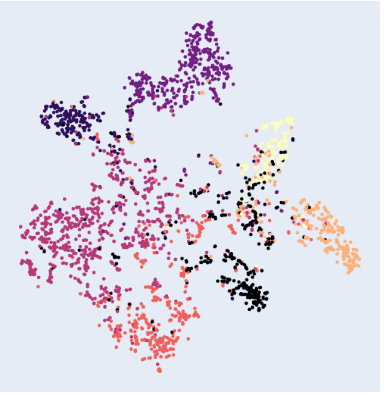
Similarly, The figure below illustrates the performance of the GAT model on the **CiteSeer** dataset when trained with **Quantization Aware Training (QAT)**, achieving an accuracy of approximately **0.700**, which is nearly equivalent to the baseline accuracy of **0.705** obtained without quantization. This demonstrates the **model's resilience** to quantization and the effectiveness of QAT in maintaining classification performance:

```
[55] test_accuracy = buildGATModel.test(visualization_2D=False, visualization_3D=False)
     print(f'Test Accuracy with Quantization Aware Training (QAT): {test_accuracy}')

Test Accuracy with Quantization Aware Training (QAT): 0.7
```

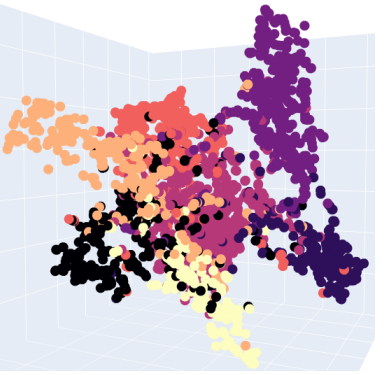
Figure 6.8: Accuracy after training model - with **Quantization Aware Training** on **CiteSeer** dataset

VISUALIZATION RESULT CLASSIFICATION

Figures 6.9 and 6.10 show 2D and 3D visualizations of the **Cora** and **CiteSeer** training datasets after applying **Quantization Aware Training (QAT)**. The node embeddings **maintain a structure** comparable to the full-precision baseline, indicating that **classification performance is preserved**. The model retains **high accuracy**, demonstrating its robustness under quantization.

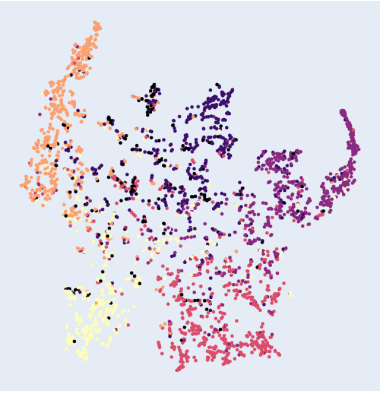


(a) **Cora** dataset: 2-D Visualization **(Quantization Aware Training)**

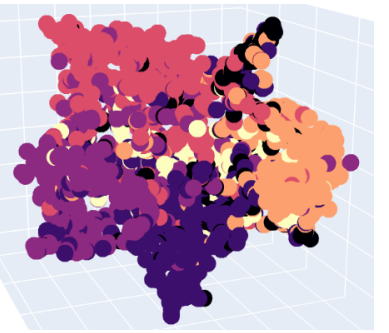


(b) **Cora** dataset: 3-D Visualization **(Quantization Aware Training)**

Figure 6.9: Visualization of classification results on the **Cora** dataset **(Quantization Aware Training)**



(a) **CiteSeer** dataset: 2D Visualization (Quantization Aware Training)



(b) **CiteSeer** dataset: 3-D Visualization (Quantization Aware Training)

Figure 6.10: Visualization of classification results on the **CiteSeer** dataset **(Quantization Aware Training)**

6.2. SIMULATION RESULT

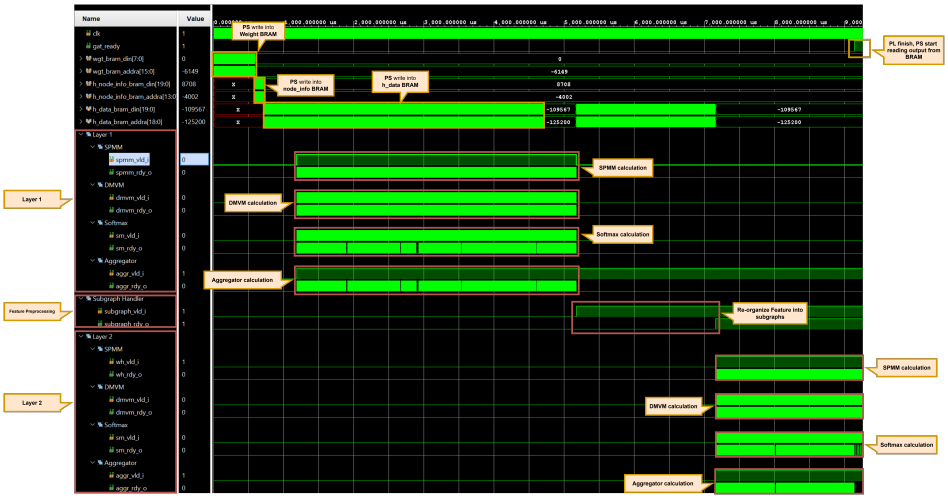


Figure 6.11: System simulation.

The simulation shown in Figure 6.11 illustrates the complete execution flow of the PL for the proposed GAT accelerator. At the start, PS loads the input matrices, including the weight matrix, feature matrix, node information, and subgraph index into the corresponding BRAMs. After memory initialization, PL begins executing Layer 1, starting with the SPMM module, followed by DMVM, Softmax, and Aggregator modules in sequence. Once Layer 1 computation finishes, PS retrieves the output features, reorganizes them into subgraphs, and writes the updated feature matrix back into the BRAMs for the next layer.

Layer 2 computation follows the same execution order, beginning with SPMM and proceeding through DMVM, Softmax, and Aggregator. The modules operate in a pipelined manner, where once the output of a module is ready, the next module immediately begins its execution without additional waiting time. This execution flow minimizes idle periods between modules. After completing all computations for Layer 2, PS can read the final output features from the BRAMs.

6.3. EXPERIMENTAL SETUP

In this chapter, we present our experimental setup to evaluate the performance and inference capabilities of our FPGA-based accelerator for Graph Attention Networks (GAT). We deploy our system on the ZCU106 board, chosen for its large BRAM capacity, which is critical for handling the memory-intensive operations of GAT, such as storing input, output matrices, and intermediate results between layers.

The ZCU106 board is equipped with a Zynq UltraScale+ MPSoC XCZU7EV-2FFVC1156, featuring 230400 LUTs, 460800 Flip-Flops, and 312 BRAMs on the PL side, enabling support for large-scale hardware designs. The PS includes a Quad-core Arm Cortex-A53 MPCore running up to 1.5GHz, suitable for coordinating control logic and interfacing with the hardware accelerator.

We use SystemVerilog to implement all computation steps of the GAT model in the PL, including Linear Transformation, Attention Mechanism, Softmax, and Aggregation. The PS is responsible for preprocessing and managing data transfers through DMA to the PL, where the core acceleration takes place.

Our primary objective is to evaluate how hardware acceleration impacts inference latency and resource utilization in comparison to a software-only implementation. Although our current design accelerates only the core GAT computations in hardware, the architecture can be extended to support additional layers or multiple heads as needed. Figure 6.12 illustrates the block diagram of the ZCU106-based implementation.

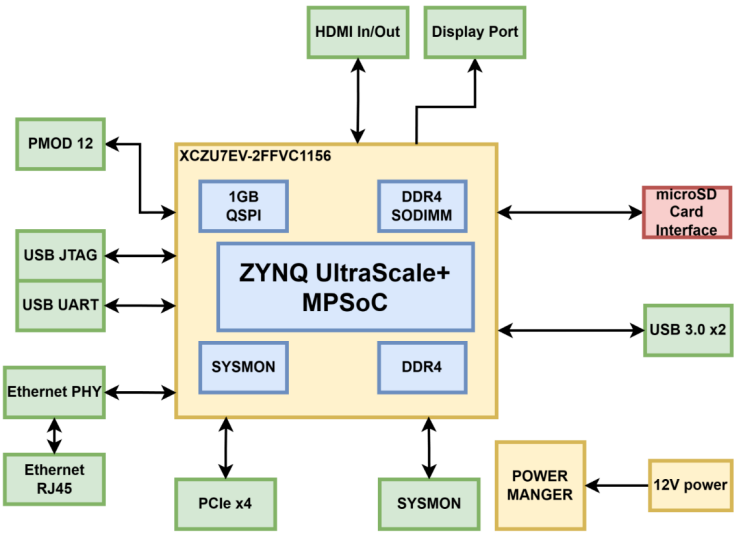


Figure 6.12: ZCU106 Block Diagram

We evaluate the performance of our system by running workloads on different hardware platforms:

- **Software on CPU and GPU:** The software implementation is tested on an different CPUs and NVIDIA GPUs, using the PyTorch library. This configuration serves as a baseline for comparing inference time and resource efficiency against the FPGA-based accelerator.

- **Accelerator on FPGA:** Our hardware accelerator is deployed on the ZCU106 board. All core GAT computations are fully implemented in PL. PS handles data preprocessing and controls the accelerator via a memory-mapped Register Bank as illustrated in Figure 3.1. Input matrices, including feature, weight, and attention weights are initially loaded into DDR4 memory, accessed by the PS, and transferred to the PL through DMA. For testing and evaluation, we run inference with two different datasets: **Cora** and **CiteSeer**. The following sections present the detailed experimental results.

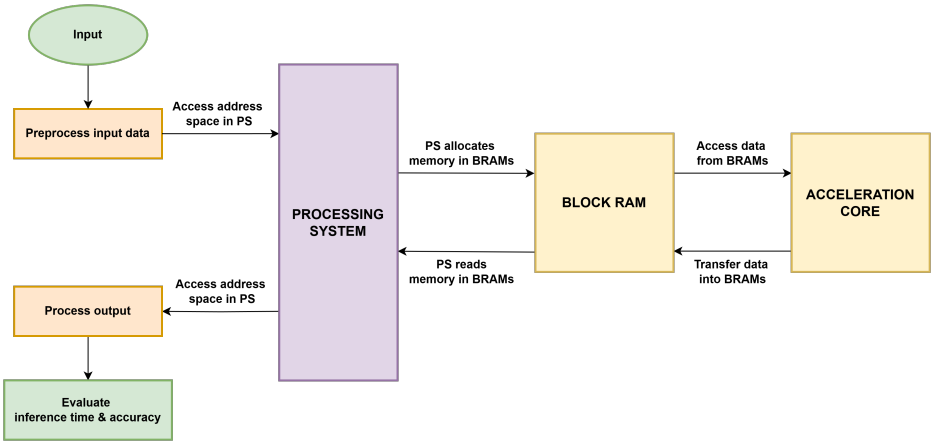


Figure 6.13: A flow diagram illustrates the overall system operation

To balance both accuracy and performance, we design our accelerator using two different approaches:

- **LayerBreak-GAT:** After completing the first GAT layer, the resulting feature matrix is transferred to the PS, where it is quantized and reorganized into subgraphs. The preprocessed features are then written back into the **H_DATA BRAM** to continue computation for the second layer.

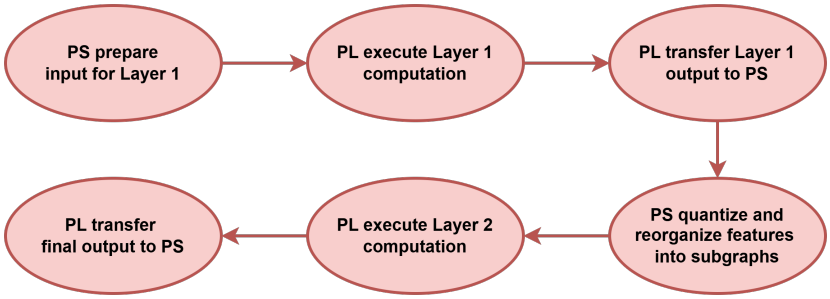


Figure 6.14: **LayerBreak-GAT** flow diagram

- **Streamline-GAT:** The entire two-layer GAT computation is executed within the accelerator without intermediate quantization. The final feature output is returned directly to the PS after both layers are processed.

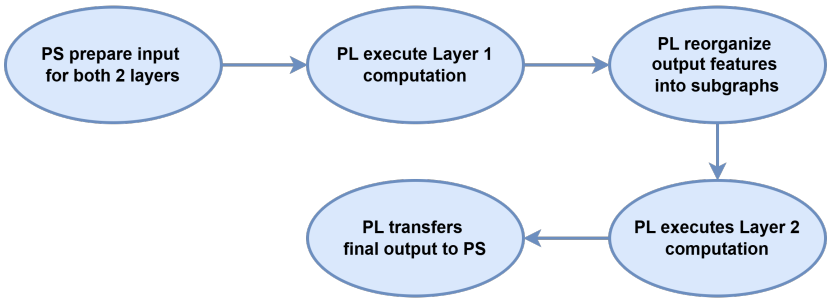


Figure 6.15: **Streamline-GAT** flow diagram

While **LayerBreak-GAT** may yield higher accuracy due to layer-wise quantization, it introduces significant overhead from frequent data transfers, quantization steps, and memory writes between PS and PL. In contrast, **Streamline-GAT** reduces latency by keeping all computations within the PL, eliminating the need for mid-layer communication. We evaluate both approaches and report their trade-offs in terms of accuracy, runtime, and resource efficiency.

6.4. SYNTHESIS RESULT

Table 6.1 presents the hardware resource utilization of the design synthesized on the Zynq UltraScale+ MPSoC ZCU106 platform. Our system achieves a stable operating frequency of 225 MHz, making it suitable for timing-critical applications.

Table 6.1: Hardware resources on ZCU106

LUTs	LUTRAMs	FFs	BRAMs	URAMs	DSPs	Frequency
64486	1986	150633	270.5	13	94	225 MHz
27.99%	1.95%	32.69%	86.70%	13.54%	5.44%	

The design consumes 27.99% of available LUTs and 32.69% of Flip-Flops, indicating a moderately complex logic structure with a balanced use of sequential elements. BRAM usage is relatively high at 86.70%, which is expected due to the architecture's reliance on on-chip memory for storing feature vectors, intermediate results, and features across layers.

LUTRAM and URAM are used at 1.95% and 13.54%, respectively. This suggests that while temporary and large-scale data buffers are required, the current configuration remains within a safe utilization range. DSP usage is low at only 5.44%, which aligns with the design's strategy of replacing arithmetic-heavy operations with shift-based logic to conserve DSP resources.

This breakdown highlights the design's efficient resource distribution, emphasizing memory throughput and logic parallelism—two essential factors for accelerating GAT workloads on FPGA.

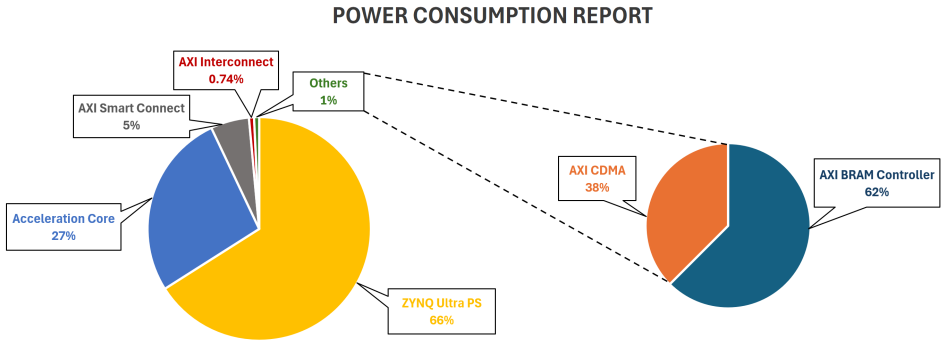


Figure 6.16: Power Consumption report

Figure 6.16 shows the dynamic power distribution of the proposed accelerator. The **Zynq Ultra PS** accounts for the largest share at **66%**, reflecting its critical role in system control and communication. **GAT Acceleration Core** consumes **27%** of the total power, highlighting the computational efficiency of the GAT engine. Minor components such as the **AXI Smart Connect** and **AXI Interconnect** contribute **5%** and **0.74%**, respectively. Within memory-related operations, the **AXI BRAM Controller** dominates at **62%**, while the **AXI CDMA** accounts for **38%**.

Overall, the results indicate that while computation is a major consumer, system and memory management still contribute significantly to total power, underlining the need for optimizing both processing and data movement.

6.5. PERFORMANCE VALIDATION AND ANALYSIS

6.5.1. COMPARISON BETWEEN APPROACHES

To evaluate the effectiveness of our two proposed architectures: **LayerBreak-GAT** and **Streamline-GAT**, we compare their performance across three key metrics: data transfer time, execution time, and inference accuracy. These metrics offer a comprehensive view of how each approach balances communication overhead, computational latency, and model accuracy. In particular, this comparison emphasizes the trade-off between introducing PS–PL interaction in **LayerBreak-GAT** and enabling fully pipelined, uninterrupted execution in the PL with **Streamline-GAT**.

TRANSFERRING TIME

Table 6.2: Transferring Time Comparison of Two Graph Attention Network Approaches

Datasets	**LayerBreak-GAT Transferring Time**		**Streamline-GAT Transferring Time**	
	Layer 1	**Layer 2**	**Layer 1**	**Layer 2**
Cora	8.96 (ms)	1.46 (ms)	9.18 (ms)	-
CiteSeer	9.84 (ms)	1.40 (ms)	10.17 (ms)	-

Table 6.2 presents the data transfer time between the PS and PL for both approaches on the Cora and CiteSeer datasets. Due to the intermediate quantization and subgraph reorganization handled by the PS, **LayerBreak-GAT** involves two data transfers: one after Layer 1 and another before Layer 2. This results in a total transfer time of **10.42 ms** for Cora and **11.24 ms** for CiteSeer.

In comparison, **Streamline-GAT** performs the entire computation within the PL and requires only a single transfer before Layer 1. While its initial transfer time is slightly higher, with an increase of approximately **2.5%** for Cora and **3.35%** for CiteSeer relative to LayerBreak-GAT, it eliminates the need for mid-layer communication and significantly reduces the overall transfer latency.

EXECUTION TIME

Table 6.3: Execution Time Comparison of Two Graph Attention Network Approaches

Datasets	LayerBreak-GAT Execution Time	Streamline-GAT Execution Time
Cora	2.24 (ms)	2.86 (ms)
CiteSeer	3.07 (ms)	3.64 (ms)

Table 6.3 presents the execution time comparison between the two approaches. **LayerBreak-GAT** completes computation in **2.24 ms** for Cora and **3.07 ms** for CiteSeer. Meanwhile, **Streamline-GAT** takes slightly longer, completing in **2.86 ms** for Cora and **3.64 ms** for CiteSeer, corresponding to an increase of approximately **27.7%** and **18.6%**, respectively. The higher execution time in **Streamline-GAT** is expected, as it performs continuous processing across both layers without interruption. In contrast, **LayerBreak-GAT** benefits from an intermediate offloading point after Layer 1, which shortens per-layer computation time but at the cost of introducing additional communication overhead elsewhere.

ACCURACY

Table 6.4: Accuracy Comparison of Two Graph Attention Network Approaches

	Datasets	
	Cora	CiteSeer
Full Precision	83.0%	72.5%
Trained Result	80.5%	70.0%
LayerBreak-GAT	80.7%	69.9%
Streamline-GAT	80.6%	69.7%

Accuracy results are shown in Table 6.4. **LayerBreak-GAT** achieves **80.7%** accuracy on Cora and **69.9%** on CiteSeer, while **Streamline-GAT** delivers slightly lower results at **80.6%** and **69.7%**, respectively. Both approaches are closely aligned with the trained quantized model and show minimal deviation from the full-precision baseline, indicating that quantization and architectural changes introduce negligible accuracy loss.

OVERALL TRADE-OFF

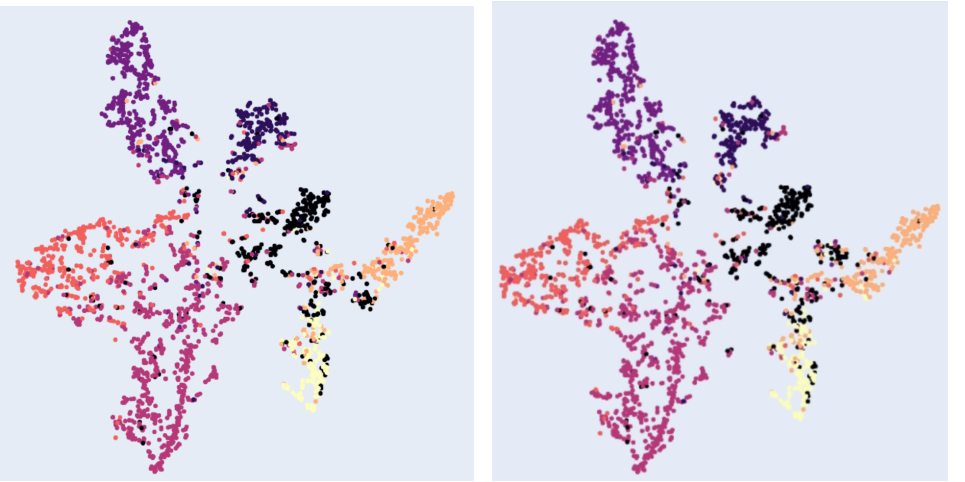
When considering data transfer time, execution time, and inference accuracy, **Streamline-GAT** demonstrates superior overall efficiency. Although **Layer**

Break-GAT achieves slightly higher accuracy and faster execution time, it comes at the cost of additional data transfer overhead and increased system complexity. The need for mid-layer communication between the PS and PL introduces significant latency penalties that offset its computational advantage. In contrast, **Streamline-GAT** maintains a fully pipelined execution within the PL, eliminating intermediate data transfers, simplifying control flow, and enabling a more consistent and scalable runtime.

In summary, while **LayerBreak-GAT** offers marginal gains in accuracy and computation speed, **Streamline-GAT** delivers a better trade-off between latency, simplicity, and practical deployment, making it the preferred solution for real-time and latency-sensitive GAT inference on FPGA.

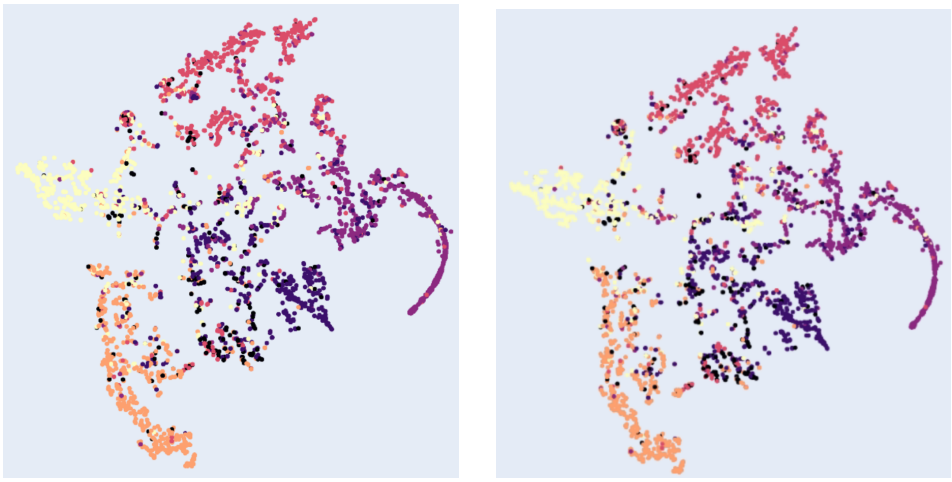
RESULTS VISUALIZATION

Figures 6.17 and 6.18 illustrate the comparison between **LayerBreak-GAT** (left) and **Streamline-GAT** (right) on the Cora and CiteSeer datasets, respectively.



(a) **LayerBreak-GAT:** Cora Dataset Visualization (b) **Streamline-GAT:** Cora Dataset Visualization

Figure 6.17: Result visualization of **Cora** dataset of two approaches.



(a) **LayerBreak-GAT:** CiteSeer Dataset Visualization (b) **Streamline-GAT:** CiteSeer Dataset Visualization

Figure 6.18: Result visualization of **CiteSeer** Dataset of two approaches.

6.5.2. COMPARISON WITH CPU AND GPU

To evaluate the scalability and efficiency of our design, we compare its performance across a range of CPU and GPU platforms, representing various levels of computing power and deployment scenarios:

- **Intel Core I5-1335U CPU @ 1.30GHz:** A mainstream laptop processor offering solid performance for standard workloads. It serves as a mid-tier benchmark to assess how our architecture compares in everyday computing environments.
- **AMD Ryzen 9 6900HS CPU @ 3.30GHz:** A high-performance processor used in gaming and creator laptops. This CPU helps evaluate the scalability and upper-bound efficiency of our architecture under high-end computing conditions.
- **ARM Cortex-A53 CPU @ 1.17GHz:** Widely used in embedded and mobile devices, this low-power processor allows us to measure how well our design performs in resource-constrained, power-sensitive environments.
- **NVIDIA GeForce MX450 GPU @ 3.2GHz:** An entry-level discrete GPU typically found in ultrabooks. It provides a baseline for GPU acceleration and highlights how our FPGA implementation compares with lightweight GPU solutions.

- **NVIDIA GeForce RTX 3050 GPU @ 3.2GHz:** A modern mid-range GPU designed for gaming and general-purpose computation. It serves as a reference for performance against more powerful GPU-based acceleration.

Table 6.5: Execution Time for Graph Attention Network across different platforms

Platform / Datasets	Cora	CiteSeer
FPGA (Ours)		
LayerBreak-GAT @ 225 MHz	2.24 (ms)	3.07 (ms)
Streamline-GAT @ 225 MHz	2.86 (ms)	3.64 (ms)
CPU		
Intel Core I5-1335U @ 1.30 GHz	158 (ms)	168 (ms)
AMD Ryzen 9 6900HS @ 3.30 GHz	175 (ms)	231 (ms)
ARM Cortex-A53 @ 1.17 GHz	865 (ms)	1173 (ms)
GPU		
NVIDIA GeForce MX450 @ 3.2 GHz	11.69 (ms)	15.53 (ms)
NVIDIA GeForce RTX 3050 @ 3.2 GHz	11.61 (ms)	13.34 (ms)

The inference time across different platforms was measured and compared to evaluate the effectiveness of the FPGA accelerator. As shown in Table 6.5, the FPGA achieves substantial reductions in inference latency compared to both CPU and GPU baselines.

Compared to CPU platforms, the FPGA accelerator achieves approximately 60 to 80 times speedup relative to the Intel Core i5-1335U and AMD Ryzen 9 6900HS, and more than 380 times speedup compared to the ARM Cortex-A53. This large improvement highlights the inefficiency of general-purpose CPUs in handling sparse, memory-bound GNN workloads, where irregular data access patterns and limited parallelism severely impact performance. In contrast, the FPGA efficiently overlaps computation and memory access through customized pipelining and sparsity-aware processing, eliminating most software overhead.

When compared to the GPU baseline, the FPGA still achieves a 4 to 5 times reduction in inference time. Despite the GPU's high theoretical throughput, its performance is less effective on irregular and sparse computations characteristic of GAT models. Meanwhile, the FPGA's fine-grained control over dataflow and fully dedicated pipelines maintain consistently low latency, even at a much lower operating frequency (225 MHz versus 3.2 GHz).

6.6. STATE-OF-THE-ART COMPARISONS

Table 6.6: Comparison of different FPGA implementations

	FP-GNN [6]	FTW-GAT [1]	SH-GAT [8]	Ours
Platform	VCU128	VCU128	AlveoU280	ZCU106
Frequency	225 MHz	225 MHz	225 MHz	225 MHz
Resource Utilization				
LUT	1068 K (82%)	437 K (33%)	110 K (8%)	151 K (33%)
FF	727 K (28%)	470 K (18%)	125 K (5%)	65 K (28%)
BRAM	1792 (89%)	1503 (75%)	1428 (71%)	270 (87%)
DSP	8704 (96%)	1216 (13%)	732 (8%)	94 (5%)
Dynamic Power (W)				
	14.8	7.62	-	4.16
Latency [Speedup]				
CR	46.3 μs [62×]	44.9 μs [64×]	19.4 μs [148×]	2.86 ms [1×]
CS	71.4 μs [51×]	50.8 μs [72×]	22.1 μs [165×]	3.64 ms [1×]
Energy Efficiency [Graph/kJ]				
CR	1.46E6 [17×]	3.65E6 [43×]	-	8.58E4 [1×]
CS	9.46E5 [14×]	3.14E6 [47×]	-	6.74E4 [1×]

The comparison results in Table 6.6 demonstrate that our proposed GAT accelerator achieves good resource efficiency and low dynamic power consumption compared to other FPGA-based GAT designs. However, the overall execution latency and energy efficiency are lower than state-of-the-art accelerators implemented on larger FPGA platforms such as VCU128 and AlveoU280. The primary limiting factor is the relatively small number of available BRAMs on the ZCU106 device.

In our design, each subgraph can be handled and computed independently, enabling the potential for parallel execution of multiple subgraphs. However, due to the limited BRAM resources on ZCU106, **h_data BRAM** can only provide a single read port, which restricts concurrent access to feature data. As a result, even though the architecture inherently supports parallel subgraph computation, in practice, subgraphs must be processed sequentially, leading to increased latency.

In contrast, designs such as FP-GNN, FTW-GAT, and SH-GAT leverage larger FPGA devices with abundant BRAM capacity, allowing them to store multiple subgraphs and perform wider parallel feature aggregation. This capability directly reduces their execution time and improves throughput. If our design were implemented on a platform with a larger number of BRAMs, it would be possible to enable true parallel processing of subgraphs, significantly accelerating the overall computation and improving energy efficiency.

Therefore, the observed performance gap is primarily attributed to hardware resource limitations rather than architectural inefficiency. Our design remains scalable, and its performance can be greatly enhanced on higher-end FPGAs with expanded BRAM resources.

7

CONCLUSION

This project successfully implements an FPGA-based accelerator for the Graph Attention Network (GAT) model, optimizing both computation efficiency and resource utilization. The architecture utilizes custom-designed pipelines for key operations such as Sparse Matrix-Vector Multiplication (SPMM), Dense Matrix-Vector Multiplication (DMVM), Softmax normalization, and Aggregation. By leveraging FPGA hardware, the design maximizes performance through efficient handling of sparsity and irregular memory access patterns in GNN workloads.

Evaluation on benchmark datasets like Cora and CiteSeer shows significant speedup in inference times. The FPGA design achieves 60-80× faster performance than CPUs and 4-5× faster than GPUs, despite operating at lower clock frequencies. These results validate the advantage of FPGA-based solutions for real-time AI applications, confirming their effectiveness in accelerating sparse data processing.

In addition to performance improvements, the design optimizes resource utilization, minimizing memory overhead and efficiently processing sparse feature matrices. This makes it particularly suitable for deployment in edge environments, where computational and energy constraints are crucial. The system's ability to operate efficiently under limited memory resources offers a promising solution for energy-conscious applications.

However, there are still limitations to address. The current design mainly supports GAT models, with limited flexibility for other GNN variants. The Subgraph Dispatcher step, which introduces some latency, requires further optimization. Additionally, while quantization and fixed-point arithmetic improve memory efficiency, they may reduce accuracy in some cases.

Future work will focus on extending the design to support a broader range of GNN models and optimizing memory access patterns and subgraph handling mechanisms. Additionally, dynamic precision adjustment and integration into edge-AI systems will improve both performance and energy efficiency. The goal is to optimize the system for practical deployment, ensuring scalability across different AI applications.

In conclusion, the FPGA-based GAT accelerator demonstrates the promising potential of hardware specialization for GNN inference, offering an efficient, scalable, and energy-efficient solution for edge computing environments.

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] Teng Tian et al. “FP-GNN: Adaptive FPGA accelerator for Graph Neural Networks”. In: *Future Generation Computer Systems* 136 (2022), pp. 294–310. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2022.06.010. URL: https://www.sciencedirect.com/science/article/pii/S0167739X22002217.

[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] Xilinx. *Programming View of Zynq UltraScale MPSoCs*. https://docs.amd.com/r/en-US/ug1137-zynq-ultrascale-mpsoc-swdev/Programming-View-of-Zynq-UltraScale-MPSoCs. 2024. URL: https://docs.amd.com/r/en-US/ug1137-zynq-ultrascale-mpsoc-swdev/Programming-View-of-Zynq-UltraScale-MPSoCs.

[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.