Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL & ELECTRONIC ENGINEERING

Bit-Level Manipulated Graph Neural Networks

Author:
Pedro Gimenes
CID: 01730633

Supervisors:
Dr Yiren Zhao
Dr Erwei Wang
Prof. George Constantinides

Submitted in partial fulfillment of the requirements for the MEng degree in Electrical & Electronics Engineering of Imperial College London

June 2023

Acknowledgements

I stand at the end of this significant academic journey with a sense of gratitude for those who have been instrumental in its accomplishment. First and foremost, I wish to express my appreciation to my supervisors, Aaron, Erwei and George, whose immense knowledge, profound experience, and thoughtful feedback have been invaluable to my learning and the development of this dissertation.

To my parents, Maria Clara and Marcelo, who have instilled in me the virtues of hard work and dedication, I owe a debt of gratitude. This achievement is as much yours as it is mine.

The journey through this MEng degree would not have been the same without the support and patience of my partner Louise, whose companionship made this journey easier and more rewarding.

Finally, I would like to express my gratitude to my fellow students and faculty who have contributed, directly or indirectly, to the successful completion of this project.

Plagirarism Statement

I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link.

I affirm that I have submitted, or will submit, an identical electronic copy of my final year project to the provided Blackboard module for Plagiarism checking.

I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me but is represented as my own work. I have not used ChatGPT4 or other Large Language Models as an aid in the preparation of my report.

Abstract

Neural networks have been widely deployed to achieve state-of-the-art performance in classification and regression tasks within various domains [1, 2]. Inference is typically performed on GPU devices, which are readily available and offer large performance improvements over general-purpose CPUs due to their deeply parallelized architecture.

As cutting-edge models become increasingly complex, GPUs have shown a performance limitation due to expensive data copy and synchronization mechanisms. In particular, extreme low-latency applications such as in high-energy physics [3] or autonomous vehicles [4] show the need for custom hardware to achieve sub-μs predictions. FPGA devices are well capable of meeting these requirements due to their reconfigurable logic fabric and have been shown to achieve up to $10\times$ latency and throughput improvements over GPU counterparts, with orders of magnitude lower power consumption [5]. The added flexibility of FPGAs, stemming from their reconfigurability, enables finer-grained optimizations in network implementation, such as layer-specific quantization schemes. This use of lower precision numerical formats has been shown to reduce memory requirements and computational overhead, as well as power consumption on reconfigurable devices, at a low cost to model accuracy [6].

In recent times, Graph Neural Networks (GNNs) have attracted great attention due to their classification performance on non-Euclidean data [7, 8]. FPGA acceleration proves particularly beneficial for GNNs given their irregular memory access patterns, resulting from the sparse structure of graphs. These unique compute requirements have been addressed by several FPGA and ASIC accelerators, such as HyGCN [9] and GenGNN [10].

Despite the relative success of hardware approaches to accelerate GNN inference on FPGA devices, previous works have shown to be limited to small graphs with up to $20k$ nodes, such as Cora, Citeseer and Pubmed. Since the computational overhead of GNN inference grows with increasing graph size, current accelerators are not well-prepared to handle medium to large-scale graphs, particularly in real-time applications.

This work introduces **AGILE** (**A**ccelerated **G**raph **I**nference **L**ogic **E**ngine), an FPGA accelerator aimed at enabling real-time GNN inference for large graphs by exploring a range of hardware optimisations. A new asynchronous programming model is formulated, which reduces pipeline gaps by addressing the non-uniform distribution in node degrees. Inspired by GNN quantisation analysis from Taylor *et al.* [11], a multi-precision node dataflow is proposed, improving throughput and device resource usage. Finally, an on-chip Prefetcher unit was implemented to cut down memory access latency. Evaluation on Planetoid graphs shows up to 2.8x speed-up against GPU counterparts.

Contents

1 Introduction

Graphs serve as powerful representations for capturing relationships between entities. In a graph, entities are represented as nodes, while their interconnections are represented as edges. This structure enables modelling a wide range of complex systems, including social networks [12], biological interactions [13], and recommendation systems [14]. Graph Neural Networks (GNNs) have emerged as a transformative approach for processing graph data, designed to learn from complex relational information by exploiting the rich interconnectedness of graphs.

An evaluation of the required hardware for Graph Neural Network acceleration begins with an analysis of its computational requirements. GNN inference can be divided into two main phases, (1) **Aggregation** and (2) **Transformation**. In the Aggregation phase, a permutation-invariant function such as summation or mean is applied over the feature embeddings of a node's neighbours. The results of this phase are then utilized in the Transformation phase, which consists of a Fully-Connected layer (i.e. multiplication with a set of learned parameters, followed by optional bias and activation) used to generate the updated feature embedding for each node. While the Transformation phase presents a highly regular computational pattern, which can be effectively accelerated on a parallelized device such as a GPU, the Aggregation phase involves many **irregular memory accesses** due to the random and sparse nature of graph data. Additionally, the latency incurred for aggregation is a function of a node's degree, which follows a highly **non-uniform distribution**. As proposed by Yan *et al.*, "*an efficiently-designed GNN accelerator needs to alleviate the computational irregularity of the Aggregation phase while leveraging the regularity of the Transformation phase*" [9].

1.1 Motivation

The performance of GNN inference on CPU and GPU devices acts as a baseline for custom accelerators. Although CPU memory systems are a well-matured and highly-optimized technology, the unpredictability of memory access patterns during GNN inference renders traditional cache systems less effective. Inference on GPU offers a large performance improvement due to the deep level of parallelism, however, these devices are still limited by **expensive memory management mechanisms**. In particular, there is no support for **intra-phase pipelining**, meaning aggregation results must be stored back into off-chip memory before being re-fetched for subsequent computation. Additionally, resource usage is not optimized for neural networks due to support for graphics workloads, and modern devices have limited support for computation with low-precision numerical formats.

These considerations have motivated the design of several custom GNN accelerators. In HyGCN (discussed in Section 2.4.1), aggregation is performed over a set of Single Instruction Multiple Data (SIMD) cores, while node transformation is performed in a **multi-granular systolic array** [9]. An on-chip sparsity reduction algorithm increases spatial locality in DRAM accesses, alleviating graph irregularity. GenGNN was proposed as a model-agnostic framework for GNN acceleration, addressing the gap between the development pace of GNN models and custom accelerators [10]. This is achieved using High-Level Synthesis tools to map models following the Message Passing Mechanism onto FPGA fabric.

Despite the benefits of previously proposed GNN accelerators over GPU counterparts, (i) the double-buffering mechanism deployed in the HyGCN programming model is not well suited for graph computation due to the non-uniform distribution of node degrees. Under this paradigm, "isolated" nodes with low neighbour counts must wait for "denser" nodes with higher neighbour counts before computation can proceed, leading to a large number of **pipeline gaps**. This highlights the need for an **asynchronous programming model**, where nodes are independently allocated resources and sched-

uled onto the accelerator. Additionally, (ii) neither accelerator offers support for **multi-precision inference**, i.e. casting sections of the computation to low-precision numerical formats. This approach has the potential to reduce resource usage or, equivalently, increase throughput/area metrics. Finally, (iii) present accelerators require on-chip buffering of node embeddings and weights. As such, these have limited applicability for inference on large graphs ($> 100k$ nodes) where embeddings cannot feasibly be stored on-chip, highlighting the need for a **streaming pre-fetching system** to hide memory access latency while the accelerator is busy.

Accelerator	Parallelization	Asynchronous Programming	Node Pre-Fetching	Multi-Precision
CPU	✗	✗	✗	✗
GPU	✓	✓	✗	✗
HyGCN [9]	✓	✗	✗	✗
GenGNN [10]	✓	✗	✓	✗
AGILE	✓	✓	✓	✓

Table 1: Summary of desirable graph processing features across available hardware platforms.

Table 1 summarizes the aforementioned aspects in the main available hardware platforms for GNN inference. Note that although CPU parallelization is possible, modern multi-threaded systems typically have a significantly limited core count compared to GPUs or other custom accelerators. Asynchronous programming is possible in GPUs at the kernel level, although computation does not follow a node-centric flow. Although pre-fetching is possible in GenGNN, this does not follow a streaming approach, meaning all incoming messages for nodes in-flight are required to be stored on-chip. The incurred resource consumption effectively reduces the potential number of parallelization channels. Finally, to the best of the author's knowledge, no presently-available accelerator supports multi-precision computation.

1.2 Contributions

This work addresses the aforementioned shortcomings by (i) exposing a set of **Nodeslots** which can be asynchronously programmed by the Host device through memory-mapped registers, where each Nodeslot is a data structure containing all required information to perform the inference pass. (ii) **AGILE** contains a heterogeneous pool of **multi-precision Aggregation and Transformation Cores**, which are dynamically allocated to Nodeslots at run-time. While previous accelerators with multi-precision support for convolutional and linear networks have defined the numerical format as a layer-wise or model-wise parameter, ***AGILE** extends this paradigm to the graph realm by defining the precision as a node-wise parameter*. The motivation for this stems from the findings of Tailor *et al.* that quantisation statistics are disproportionately skewed by nodes with large neighbour counts, as discussed in Section 2.2.1. Finally, (iii) an on-chip **Streaming Prefetcher** acts to maximise memory bandwidth at increased parallelization levels.

In summary, the main contributions of this project are as follows.

- **Asynchronous Programming Model**: enables reducing graph processing latency by leveraging the non-uniform distribution of node degrees.
- **Mixed Precision**: **AGILE** supports a number of numerical formats and the ability to specify node-wise quantization strategies, improving resource utilization with retained model accuracy.
- **Large Graph Handling**: while prior accelerators have focused on GNN inference on small-scale graphs, **AGILE** enables computation on large graphs ($> 100k$ nodes) through its on-chip Streaming Prefetcher unit.

The body of this report is structured as follows. Section 2 contains background on Graph Neural Network architectures, approaches to scale inference for large graphs and presently available ASIC and FPGA accelerators. Section 3 presents an overview of the features of **AGILE**, its programming model and performance optimizations. Section 4 presents the circuit-level description of each functional unit within the accelerator. Section 5 presents the Verification Plan and techniques used to test the required functionality. Section 6 presents the Implementation Plan, with a discussion of the requirements for each milestone over the course of the project. Section 7 presents latency and resource usage results for the evaluation of the accelerator against realistic GNN use cases. Finally, Section 8 discusses elements for future work.

2 Background

Before introducing the project, it is worth establishing an overview of its background and context. Section 2.1 provides an overview of Graph Neural Networks and some widespread model architectures supported by the presented accelerator. Section 2.4.1 discusses HyGCN, one of the earliest proposed accelerators for inference on GCN networks. Section 2.4.2 provides an overview of GenGNN, a recent accelerator targeting flexibility in supporting arbitrary model architectures.

2.1 Graph Neural Networks

A graph represents a set of objects (nodes) with connections between them (edges). Each node and edge has a feature embedding with associated data. Edges can be directed, in the case when information flows from a source node to a destination node, or undirected.

2.1.1 Applications

Three general types of prediction tasks can be achieved on graph data: graph-level, node-level and edge-level. In *graph-level tasks*, a large number of graphs with a smaller number of nodes is used in training to classify properties of unseen graphs. An example is in antibiotic discovery [15]. Here, molecules are represented as graphs, where the nodes represent atoms and the edges represent covalent bonds.

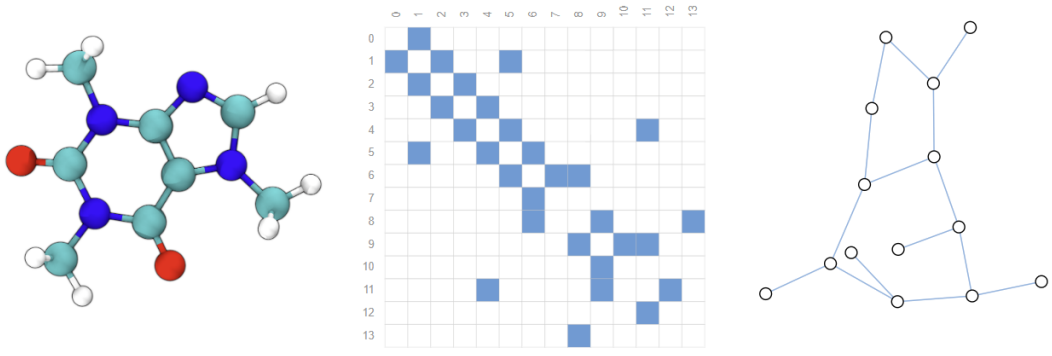

Figure 1: The 3D representation of the Caffeine molecule (left), along with its Adjacency matrix of covalent bonds between atoms (center) and graph representation (right) [16]

In *edge-level tasks*, the feature embedding of a number of nodes is used to predict the presence or feature embedding of the edges between them. This has been deployed in recommendation systems

for social network graphs, where the presence of edges indicates a potential link between two users. In *node-level tasks*, the feature representation of a node and its neighbours is used to predict information regarding that node. In citation network graphs, where links represent mentions of other works within academic papers, node-level classifiers can predict author or paper attributes, such as the field of study from neighbouring citations.

2.1.2 Graph Representation

A graph $G = (\mathcal{V}, \mathcal{E})$ is a collection of nodes/vertices $\mathcal{V}$ and edges $\mathcal{E}$. The set of feature representations is denoted by matrix $\mathbf{X} \in \mathcal{R}^{N \times D}$, where $N = |\mathcal{V}|$ is the number of nodes and D is the dimension of the node embeddings. An element $e_{i,j} = (v_i, v_j)$ present in the set $\mathcal{E}$ indicates that there is a connection between nodes v_i and v_j. In an undirected graph, the edge element $e_{i,j}$ corresponds to $e_{j,i}$. The connections in a graph can be represented using an **Adjacency Matrix**. This is a $N \times N$ matrix, where each element $A_{i,j}$ represents an edge between nodes i and j, i.e.

$$A_{i,j} = \begin{cases} 1 & e_{i,j} \in \mathcal{E} \\ 0 & e_{i,j} \notin \mathcal{E} \end{cases} \tag{1}$$

Notation	Meaning
$G(\mathcal{V}, \mathcal{E})$	A graph with associated set of nodes $\mathcal{V}$ and set of edges $\mathcal{E}$.
N	Number of nodes in the graph, i.e. $\|\mathcal{V}\|$
$\mathcal{N}_i$	Set of neighbours of node i
A	Adjacency Matrix
X^l	Feature matrix at layer l, with each row x_j^l corresponding to a single node
W	Matrix of learned parameters for node feature transformation

Table 2: Commonly used notations in representing Graph Neural Networks

One of the challenges of applying Machine Learning methods to graph data is that the Adjacency matrix is not permutation invariant, i.e. reorderings of the rows and columns correspond to the same graph, although they may have an effect on the accuracy of naive models.

2.1.3 Graph Convolutional Networks (GCN)

Graph Convolutional Networks emerged as a solution analogous to Convolutional Neural Networks in the image domain. To generalize convolutions on graphs, which form a non-regular structure, requires each node to perform a permutation-invariant aggregation of the feature embeddings of its neighbours. The update law for a single GCN layer, proposed by Kipf et al. [17], is shown in Equation 2.

$$H^{l+1} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{2}$$

Here, the diagonal degree matrix $\hat{D}$ is defined as $\hat{D}_{ii} = \sum_j \hat{A}_{ij}$. The $\hat{A}$ matrix represents the adjacency matrix with added self-connections, i.e. $\hat{A} = A + I$ where I is the unit matrix. H^l represents the output activations at layer l, such that $H^0 = X$, i.e. the input activations equal the features of the input graph. σ represents a non-linear activation function, typically $ReLU(\cdot) = max(0, \cdot)$. Finally, W^l represents a set of layer-wise learned parameters.

The multiplication of the adjacency matrix with the previous layer's activations corresponds to a sum aggregation over the neighbours of each node. The aggregated features are normalized by the square

root of the degree of each node before multiplication with the learned parameters and activation function, corresponding to a Fully-Connected layers.

A single GCN layer applied to an input graph leads to an output graph of the same size, where each node feature has learned parameters from its immediate neighbours. By stacking several GCN layers, a node's features will also be updated according to further away neighbours. A node-level prediction task can then be achieved by stacking a **Multi-Layer Perceptron**, taking a single feature embedding with Softmax activation to classify any given node.

2.1.4 Graph Attention Networks (GAT)

Attention mechanisms have become a de facto standard in predicting sequence-based data, such as in Natural Language Processing [18]. This has been shown to improve model performance by enabling classifiers to focus on more relevant information. Inspired by this, Graph Attention Networks (GAT) were proposed as an attention-based architecture for node classification on graphs.

Graph Attention Networks rely on the computation of *attention coefficients* for each node pair in a neighbourhood $\mathcal{N}$. Firstly, the e_{ij} coefficients are obtained, which express the importance of node j to node i. This is achieved by means of a Fully-Connected Layer, or Multi-Layer Perceptron, applied to the concatenated feature embeddings: $e_{ij} = a(x_i, x_j)$. As shown in Equation 3, a Softmax activation function is then applied to normalize the importance coefficients across the neighbourhood of the node.

$$\alpha_{ij} = softmax_j(e_{ij}) = \frac{exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} exp(e_{ik})} \tag{3}$$

In a GAT Layer, the update to the feature embedding of each node is then expressed as the aggregation of neighbouring nodes, weighted by the attention coefficients, as shown in Equation 4.

$$x_i^{l+1} = \sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} W x_i^l \right) \tag{4}$$

Furthermore, the Multi-Head attention mechanism can be deployed by repeating the mechanism shown in Equation 4 up to K times and concatenating the resulting embeddings.

2.2 Neural Network Quantization

Quantization has been widely explored as a method for reducing model complexity and computational latency in neural networks. Networks can benefit from low-precision numerical representations through Quantization-Aware Training (QAT), which aims to minimize accuracy loss at these low-precision representations. This is typically achieved by quantizing activations in the forward pass, making use of the Straight-Through Estimator (STE) in the backwards pass to estimate the non-differentiable quantization gradients.

In general, activations are quantized in the forward pass following Equation 5, where q_{min}, q_{max} form the representable range of floating-point values. Here, s is the scaling factor to force x into the required range and z is the zero-point, i.e. the floating point equivalent of the value 0 in the quantized space.

$$x_q = \min(q_{max}, \max(q_{min}, \left[\frac{x}{s} + z \right]) \tag{5}$$

The $\min$ and $\max$ functions are in place to show that any values beyond the specified range assume the fixed-point value at the limit. Following this, activation can be de-quantized following Equation

6. Here, $\hat{x}$ is an approximation of the original floating-point value. This approximation is better for higher bit-widths or narrower floating-point ranges.

$$\hat{x} = (x_q - x)s \tag{6}$$

2.2.1 Degree-Quant: Quantization-Aware Training for Graph Neural Networks

Degree-Quant, proposed by Tailor *et al.*, was one of the first suggested approaches in applying Quantization-Aware Training to Graph Neural Networks [11]. Firstly, Tailor *et al.* suggest that the aggregation phase of GNN inference is the predominant source of quantization error. This effect is observed more heavily in nodes with higher in-degrees, which can be intuitively explained since the absolute magnitude of aggregation grows with the number of neighbours. This causes the quantization range statistics to be affected by these outliers in the distribution, reducing the quantization resolution in the majority of cases.

An evaluation of the mean and variance of message aggregations across network layers shows that the rate of growth in aggregation with respect to node degree varies according to the model type. The worst case is Graph Isomorphism Networks (GIN), which see $O(n)$ growth, followed by GCN ($O(\sqrt{n})$) and GAT ($O(1)$). Equation 7 shows the derivative of the loss function $\mathcal{L}$ with respect to the weights for a GCN layer, where $d_{i,j}$ are the node degrees and $\mathbf{h}_{l+1}^{i}$ are feature embeddings. It can be seen that the expected value of the error in loss gradient is a function of the error in node aggregation, $\mathbf{y}_{\text{GCN}}$.

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{i=1}^{|\mathcal{V}|} \sum_{j \in \mathcal{N}_i} \frac{1}{\sqrt{d_i d_j}} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{h}_{l+1}^{i}} \circ f'(\mathbf{y}_{\text{GCN}^i}) \right) h_l^{j^T} \tag{7}$$

DegreeQuant addresses the issue of quantization error by stochastically applying a protection mask at each layer following the Bernoulli distribution. Protected nodes operate at floating-point precision, while non-protected nodes operate in fixed-point. A node's probability of protection is a function of its degree, interpolated within a parametrizable range $[p_{min}, p_{max}]$, where the graph nodes with minimum/maximum neighbour counts are assigned the limit probabilities. Additionally, fluctuations in aggregation variance are reduced by the use of percentiles, which involves clipping a fraction of the values at the upper and lower end of the ordered aggregation tensor before the estimation of quantization statistics. Tailor *et al.* show that the proposed methods lead to significantly improved accuracies for quantised models.

2.3 GraphSAINT: Graph Sampling Based Inductive Learning Method

GraphSAINT was proposed by Zeng *et al.* as a method to scale GCN to large graphs by addressing the problem of neighbour explosion. While most previous methods revolve around sampling across layers, GraphSAINT suggests constructing a full GCN after sampling the training graph. Sampling algorithms are based on collecting nodes with high influence on each other within sub-graphs, however, this introduces bias while training their embeddings due to the non-uniform sampling probability. Zeng *et al.* propose sampling algorithms formulated to minimize variance while applying normalization to reduce bias.

Recall the general node embedding update law defines in Equation 8. Here, $\tilde{A}$ is the normalized adjacency matrix and $\tilde{x}_u^{(l)} = (W^{(l)})^T x_u^{(l)}$ where W is the matrix of layer weights.

$$x_v^{l+1} = \sigma \left(\sum_{u \in \mathcal{V}} \tilde{A}_{u,v} \tilde{x}_u^{(l)} \right) \tag{8}$$

Considering a single layer of a GCN network, where nodes u are in the neighbourhood of a node v, GraphSAINT proposes a modification to the pre-activation neighbour aggregation ζ as shown in Equation 9, where $\alpha_{u,v}$ is the aggregator normalization constant. Note the indicator function $\mathbb{1}_{u|v} = 1$ only if $(u, v) \in \mathcal{E}_s$ where $\mathcal{E}_s$ is the set of sampled edges, hence the summation is performed over the sampled neighbourhood of v.

$$\zeta_v^{(l+1)} = \sum_{u \in \mathcal{V}} \frac{\tilde{A}_{u,v}}{\alpha_{u,v}} \tilde{x}_u^{(l)} \mathbb{1}_{u|v} \tag{9}$$

Zeng *et al.* show that by defining $\alpha_{u,v} = \frac{p_{u,v}}{p_v}$, where $p_{u,v}$ is the probability of sampling the (u, v) edge and p_v is the probability of the node v being sampled in the subgraph, $\zeta_v^{(l+1)}$ is an unbiased estimator of the aggregation in the non-sampled GCN. Note that these probabilities can be analytically determined for random node and edge samplers, however, they must be estimated through heuristics in the general case. This can be achieved by applying a general sampling algorithm over N iterations, keeping a count C_v and $C_{u,v}$ for each time a node and edge are selected. The normalization constant can then be taken as $\alpha_{u,v} = \frac{C_{u,v}}{V_v}$.

Additionally, Zeng *et al.* define an edge sampler to minimize the variance of aggregation estimators across all nodes and layers, ζ. An expression for this quantity can be derived with respect to the edge sampling probabilities, as shown in Equation 10. Here, the edge embedding is given by $b_e^{(l)} = \tilde{A}_{v,u} \tilde{x}_u^{(l-1)} + \tilde{A}_{u,v} \tilde{x}_v^{(l-1)}$ and the normalization factor p_v is included such that ζ is an unbiased estimator of the sum of node aggregations across all layers.

$$\zeta = \sum_l \sum_{v \in \mathcal{G}_s} \frac{\zeta_v^{(l)}}{p_v} = \sum_l \sum_{v,u} \frac{\tilde{A}_{u,v}}{p_v \alpha_{u,v}} \tilde{x}_u^{(l)} \mathbb{1}_{u|v} \mathbb{1}_v = \sum_l \sum_e \frac{b_e^{(l)}}{p_e} \mathbb{1}_e^{(l)} \tag{10}$$

Zeng *et al.* show that by constraining an edge sampling budget $\sum p_e = m$ and assuming independent edge sampling, the variance is minimized across each dimension of ζ when the edge sampling probabilities are taken as $p_e = \frac{m}{\sum_{e'} ||\sum_l b_{e'}^{(l)}||} ||\sum_l b_e^{(l)}||$. Although this requires the computation of the node embeddings in each layer, a reasonable simplification is to remove these from the estimation such that the probability depends only on the graph topology, i.e. $p_e \propto \tilde{A}_{v,u} + \tilde{A}_{u,v} = \frac{1}{deg(u)} + \frac{1}{deg(v)}$.

2.4 Graph Neural Network Accelerators

2.4.1 HyGCN

HyGCN [9] is an ASIC accelerator for inference on Graph Convolutional Networks (GCN). The authors divide GCN inference into two main phases, Aggregation and Combination. In the former, an order-invariant aggregation function is applied to the feature vectors of neighbours to a node. This operation suffers from an irregular computational pattern due to the random and sparse structure of the graph data. The latter consists of sparse Matrix-Vector Multiplication (MVM), which is a highly regular operation that can be parallelized for low latency. The implementation of dedicated Aggregation and Combination Engines results in a hybrid architecture aimed at performing inference on Graph Convolutional Networks (GCN) by alleviating the aggregation irregularity while leveraging the computational regularity of the Combination phase.

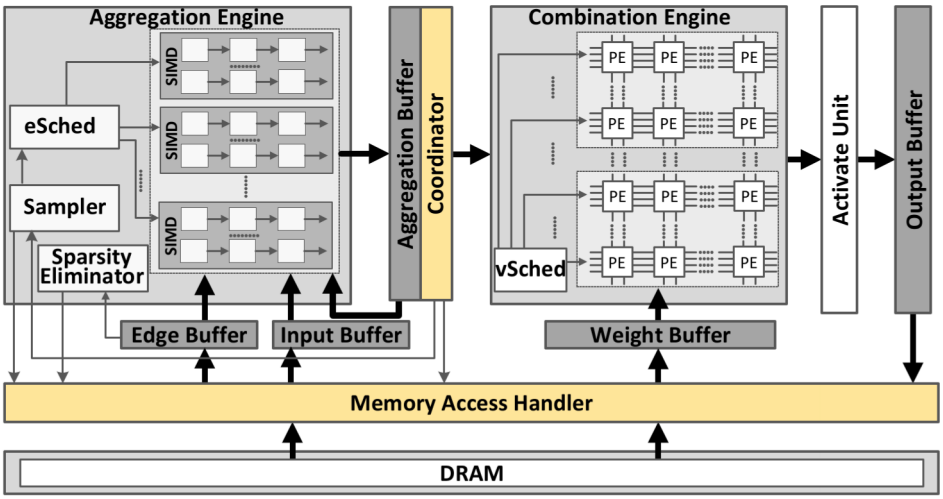

Figure 2: Microarchitecture of the HyGCN [9] accelerator. The Input, Edge and Weight Buffers are populated with node and edge embeddings and layer weights, respectively. The SIMD cores in the Aggregation Engine compute the neighbour feature aggregations, which are stored in the Aggregation Buffer. The weights are multiplied with aggregated features in the Systolic Arrays within the Combination Engine, after which a non-linear activation is applied to compute the updated node embedding. These are finally written to DRAM from the Output Buffer.

The Aggregation Engine performs node feature aggregation using a number of SIMD cores which are assigned work from an edge-centric Scheduler. Two working modes are supported, (i) vertex-concentrated and (ii) vertex-dispersed. In (i), vertex parallelism is exploited by assigning a SIMD core to aggregate features for each vertex. In (ii), intra-vertex parallelism is exploited by assigning a subset of vertex features to each SIMD core. Vertex-dispersed mode is predominantly used as this leads to lower vertex latency. Moreover, due to the wide distribution of vertex degrees, (i) leads to work imbalance since faster vertices need to wait for slower vertices.

In the Aggregation Engine, double buffers implemented with embedded DRAM are used to cache edges and neighbour feature data, as well as aggregation results before they are consumed by the Combination Engine. For each vertex, a Sampling Unit selects a number of neighbouring edges according to a pre-defined distribution, which reduces computational complexity at a low cost to model accuracy. Additionally, a dynamic sparsity elimination algorithm is implemented in hardware which acts to reduce redundant memory accesses. The graph data is partitioned in vertex interval and edge shards following [19], and subsequently, a window-based sliding-and-shrinking method is used, resulting in

denser graph partitions.

The Combination Engine performs floating point matrix multiplication using multi-granular systolic arrays [20]. Systolic modules, comprising a group of systolic arrays, can be used in either (i) independent or (ii) cooperative mode. In (i), each module processes a small group of vertices as soon as their aggregated features are ready in the Aggregation Buffer. In (ii), a larger group of aggregated features are assembled, and all systolic modules are combined such that weight parameters only propagate through the array once. Independent mode offers lower latency when coupled with the vertex-dispersed operating mode of the Aggregation Engine. However, the cooperative mode leads to lower energy consumption due to the reduced overhead of parameter sequencing.

Following matrix multiplication, an Activation Unit is used within the Combination Engine to produce the new feature vectors. Similarly to the Aggregation Engine, double buffers are employed for closer locality of weight data and output features before writing back into memory via the Memory Access Handler.

An inter-engine pipeline between the Aggregation and Combination Engines is realized by the use of a ping-pong buffering mechanism, in which the aggregation buffer is split into two regions. While the first region is read by the Combination Engine, the second is written by the Aggregation Engine with the new aggregated features.

The Memory Access Handler assigns a fixed priority to each of the main requestors to DRAM, those being the Edge and Input Buffers in the Aggregation Engine and the Weight and Output Buffers in the Combination Engine. The access priority is given by: edges $>$ input $>$ weights $>$ output. This is to leverage address continuity in DRAM requests. Furthermore, requests are processed by vertix batches, such that low-priority requests to the current batch are processed before high-priority requests to the next batch.

The authors argue that current inference accelerator counterparts are inefficient for the GCN use case. Caching techniques in modern CPUs alleviate memory fetch latency; however, they cannot account for graph irregularity, which affects the predictability of memory accesses, leading to a high MPKI rate (cache misses per kilo-instruction). While GPU inference leverages the regularity in the Combination phase, the data copy and synchronization between threads for the weight/parameter reuse are prohibitively expensive, accounting for over $\frac{1}{3}$ of execution time. Performance benchmarking of the HyGCN accelerator against Intel Xeon CPU and NVIDIA V100 GPU counterparts led to 1509X and 6.5X speed improvement, respectively.

2.4.2 GenGNN

As noted by the authors of GenGNN [10], progress in GNN models far outpaces the development of custom inference accelerators. This highlights the need for flexible accelerators capable of handling a vast array of models and datasets. Advanced GNNs such as Graph Attention Networks (GAT), Principal Neighbourhood Aggregation (PNA) and Graph Isomorphism Networks (GIN) impose additional requirements from the GCN baseline, including computation of edge embeddings and complex aggregation functions. GenGNN is a GNN inference acceleration framework developed with a focus on (i) generality and (ii) enabling real-time applications by a limited requirement for graph pre-processing.

The generalization of GNN models beyond GCN is based on the message-passing mechanism, expressed in Equation 11, which represents the node embedding update for each layer.

$$x_i^{l+1} = \gamma(x_i^l, \mathcal{A}_{j \in \mathcal{N}(i)}(\phi(x_i^l, x_j, e_{i,j}^l))) \tag{11}$$

The function ϕ represents the calculation of the message between nodes i, j, $m_{i,j}$, evaluated from the node embeddings x_i, x_j and the edge embedding at layer l, $e^l_{i,j}$. $\mathcal{A}$ represents a permutation-invariant aggregation function (such as sum, mean or standard deviation) applied to the messages from all neighbour nodes, i.e. elements of the set $\mathcal{N}(i)$. Finally, the update in the node embedding is evaluated from the γ function, which takes the concatenation or weighted sum of the current node embedding x^l_i with the aggregated message.

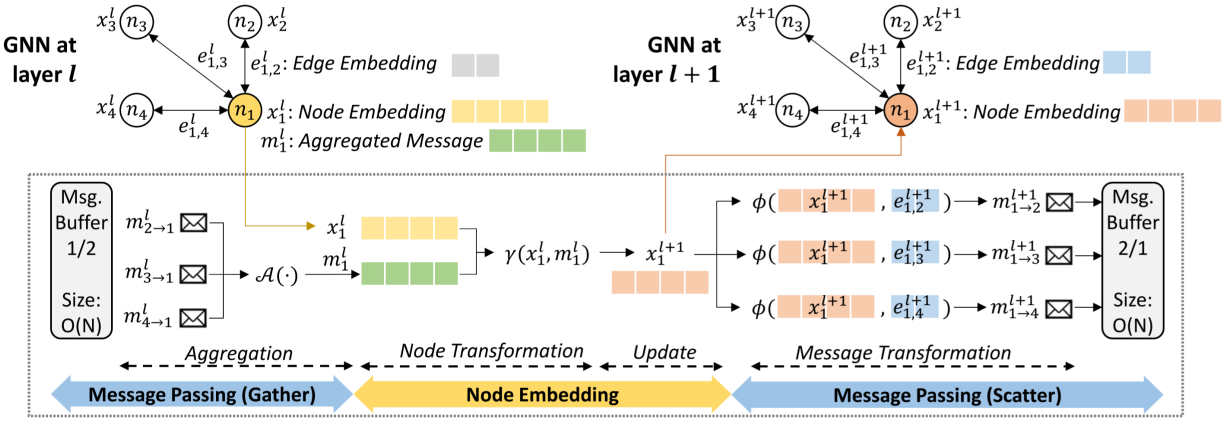

Figure 3: Computational flow of a GNN layer following the Message Passing Mechanism [10]. The Message Gather stage consists of the $\mathcal{A}$ aggregation function applied to the messages from neighbours of node n_1. In the Node Embedding stage, the aggregated messages are concatenated with the current node embedding of n_1 to compute the embedding in the next layer x^{l+1}_1. Finally, the Message Scatter stage computes the outgoing messages from the edge and updated node embeddings and sends these to subsequent layers.

Many major GNN models can be represented following the discussed paradigm. In GAT networks [21], the message scatter function ϕ takes into account the attention mechanism as shown in Equation 12. The attention coefficients $\sigma_{i,j}$, evaluated using an attention function $\sigma(i,j) = \zeta(x_i, x_j)$ such as an MLP, express the importance of node j's features to node i.

$$\hat{\phi}(x_i, x_j) = x^l_i + \sigma^l_{i,j} \cdot \phi(x^l_i, x_j) \tag{12}$$

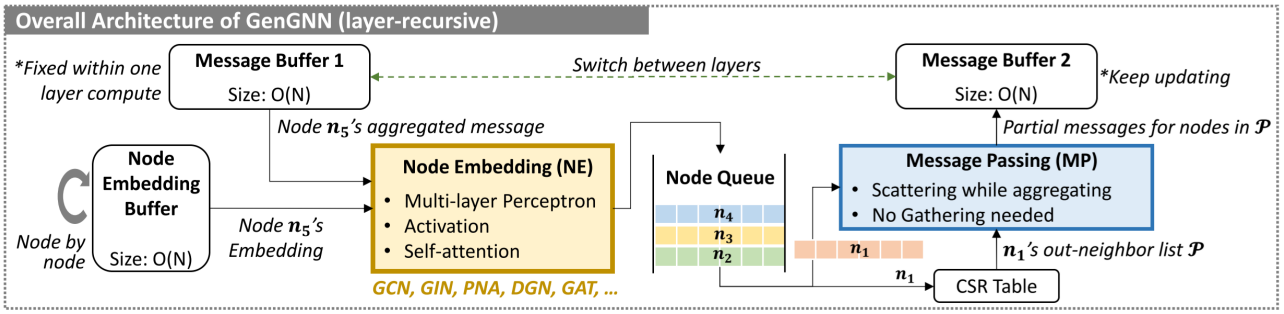

Figure 4: Architecture of the GenGNN accelerator [10], repeated for each layer of the model. A double buffer is used for the node messages such that the Node Embedding and Message Passing Processing Elements are both active. The PEs are pipelined via a Node Queue. The neighbour nodes are obtained by the MP PE from the graph data stored in CSR format.

GenGNN is implemented using a High-Level-Synthesis flow. Figure 3 shows how the Message Passing mechanism is mapped onto Hardware. The processing is split among the Node Embedding and the Message Passing Processing Elements.

3 Architectural Specification

This section contains a high-level architectural description of **AGILE**, its core functionality and its interface with Host Software.

3.1 Top-Level Overview

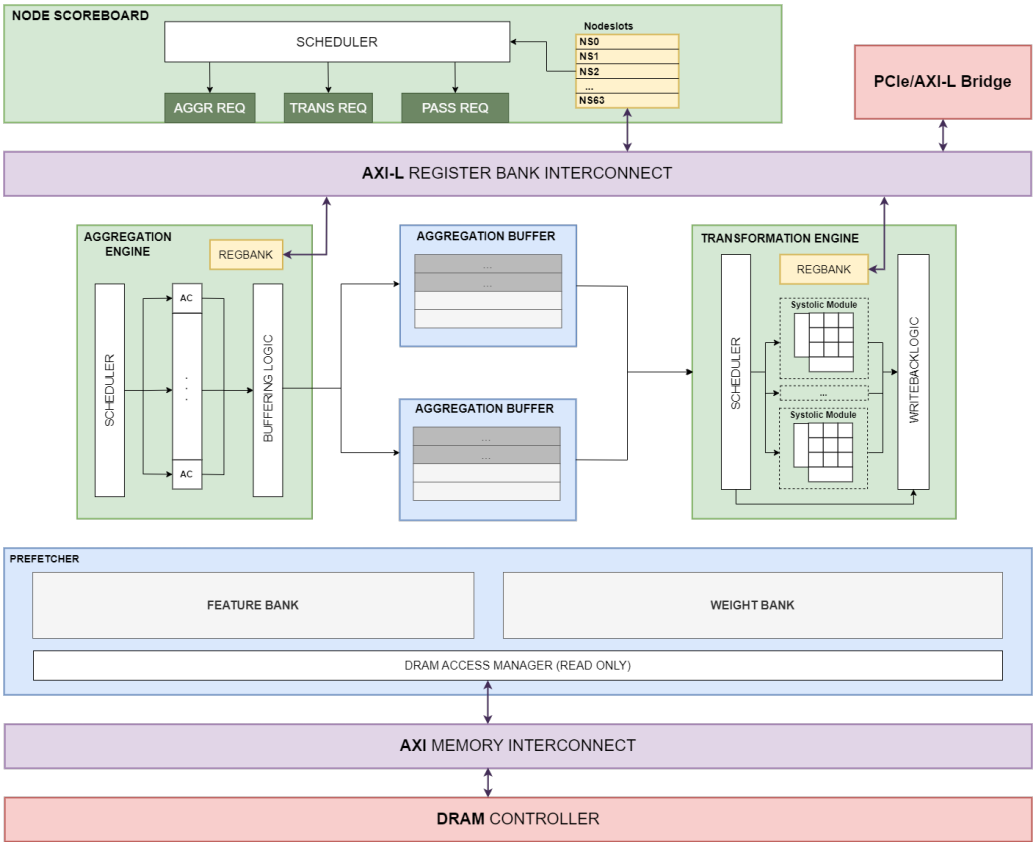

Figure 5: AGILE Top Level Diagram. The main functional units are shown in green. Blue blocks represent various memories responsible for storing graph data and intermediate computations. Purple blocks represent AXI interconnects for Host and Memory communication. Xilinx IP blocks are shown in red.

As shown in Figure 5, **AGILE** is composed of the following functional units.

- **Node Scoreboard (NSB)**: responsible for communication with the Host device and scheduling workload onto the accelerator. The Host issues instructions to **AGILE** through the AXI-Lite interconnect. Firstly, layer configuration is programmed to determine global parameters used by the accelerator. Subsequently, node information is programmed into Nodeslots, which maintain a state machine for each node currently in-flight within the accelerator.
- **Aggregation Engine (AGE)**: responsible for performing permutation-invariant aggregation functions over all neighbours of a node. After receiving an aggregation request from the NSB for an arbitrary node, AGG requests node features from the **Feature Prefetcher** and assigns the node to a subset of its **Aggregation Cores (AGC)** according to the required numerical precision. Subsequently, aggregation results are stored in the **Aggregation Buffer**.
- **Prefetcher (PREF)**: at the start of layer computation, the Prefetcher receives a request from the NSB to fetch the weights required for updating the node feature embeddings from DRAM

through its AXI interface. This operation takes place once per layer. Subsequently, the Prefetcher fetches neighbouring features and edge embeddings for each node in the graph. This is executed while the Aggregation and Transformation Engines are fully populated with work to hide the latency of memory access. Weights, features and edges are stored in local RAMs until required by subsequent steps in the pipeline.

- **Feature Transformation Engine (FTE)**: whenever aggregation results are available in the Aggregation Buffer, the FTE computes the updated feature embeddings by performing matrix multiplication with the learned parameters stored in the **Prefetcher**. Results are subsequently written back out to DRAM or optionally written into the Transformation Buffer for further processing.

The AXI-L interconnect links the host to several register banks within the design for configuration purposes. An AXI interconnect links the DRAM Controller to the Prefetcher for access to external memory. See Section 6.1 for further discussion on AXI interconnects and other Xilinx IPs.

3.2 Asynchronous Programming through the Node Scoreboard

The Node Scoreboard (NSB) is the primary interface between the Host and **AGILE**. To perform inference on a Graph, the Host must issue instructions into the accelerator via a sequence of register writes to the register bank in the NSB. The NSB decodes these instructions and drives functional units within the Microarchitecture to fetch feature data from memory and perform node aggregation and transformation before storing the results back into DRAM.

At the start of operation, the Node Scoreboard must be configured by the Host with a set of graph and layer parameters such as the total number of nodes and edges. Crucially, the input and output feature counts for each layer are required to determine the address range for each node in the memory storage elements (Prefetcher and Aggregation/Transformation/Output Buffers). Once the graph/layer configuration is marked as valid, the Host is free to start programming Nodeslots into the scoreboard.

3.2.1 Node Slots

The NSB keeps a state machine for each node being processed by the accelerator in its scoreboard, which is a register bank containing up to 64 Node Slots (indexed by a 6-bit Slot ID). Each Node Slot (NS) contains all the information required by **AGILE** to compute the node aggregation and transformation. Based on these payloads, the Node Scoreboard drives the internal functional units to perform the computation and updates its internal state machine after each functional step. As such, no further intervention is required from the Host after the Node Slot programming, which reduces the overhead of workload distribution on the Host. When the computation of a node is complete, the NSB sends an interrupt to the Host to handle any required housekeeping.

The fields shown in Figure 6 represent a subset of the total number of fields in the NSB scoreboard. Although data transfers between functional units in the accelerator take place in reference to each node's Slot ID, the NSB also keeps a copy of the Node ID, which is a 20-bit field used to differentiate each individual node in the graph. As such, **AGILE** is capable of supporting graphs up to 1M nodes (although this number is parametrizable during configuration).

AGILE supports defining custom arithmetic types for each node in the graph through the **PRECISION** field in the Node Scoreboard. The benefits of lower precision numerical formats during neural network inference have been shown in works such as by Nagel et al. [6]. The motivation for mixed precision nodes in GNN inference lies with the observation by Tailor et al. [11] that the accuracy cost of quantization is predominantly due to the aggregation phase and is directly correlated to a node's

SLOT ID	NODE_ID	PRECISION	NODE_STATE	NEIGHBOUR_COUNT	ADJACENCY_LIST_ADDR	OUT_MESSAGES_ADDR
0	0xB9E	FLOAT_32	TRANSFORMATION	645	0x3BC90188	0x4FE8B774
1	0xB9D	FLOAT_32	AGGREGATION	345	0xCAF5C03F	0xE672109F
2	0xCE65F	INT_8	PREFETCH	132	0x7426A1D6	0x8AC73FBB
3	0x51B24	FLOAT_16	EMPTY	X	0xE7F80573	X
4	X	X	EMPTY	X	X	X
...	...	...	...	...	...	...
62	0xCE423	INT_8	PREFETCH	45	0x78E26A27	0xA4D89ED9
63	0x3A914	INT_16	PROG_DONE	55	0x6A2CE10F	0xF6569187

Figure 6: Example state of the Node Scoreboard at runtime. NS0 and 1 use floating-point activation precision due to their higher neighbour count. NS 3 and 4 are in the EMPTY state, but NS3 has already been programmed with the address for incoming messages. NS62 has received all its programming, but it's pending backpressure from the Prefetcher. Nodeslots 5 $\rightarrow$ 61 follow a similar structure.

degree. As such, quantization strategies where high-precision numerical types are used for "crowded" nodes and low-precision types are used for more isolated nodes lead to decreased inference latency at a significantly reduced cost to model accuracy.

It should be noted that the order in which nodes are programmed within the Nodeslots does not imply any priority or time correlation. Various nodes may have vastly different execution times depending on neighbour count, numerical precision etc. Whenever a Nodeslot finishes its computation, it can be immediately reprogrammed by the Host with the next node. Within the NSB, nodes are serviced with Round-Robin arbitration in each clock cycle to determine which node gets access to resources within the Aggregation and Transformation Engines.

3.2.2 Node Slot State Machine

1. **EMPTY**: in this state, the Nodeslot has no active workload on the accelerator and the Host is free to program payloads into the NSB. A mask of all empty slots is visible from the Host so that it can decide what slot to use from a single register poll.
2. **PROG_DONE**: the Nodeslot transitions into this state when a write is received to the corresponding slot in the auto-clearing **MAKE_VALID** register. In this state, the Node Scoreboard waits until the Prefetcher can receive a fetch request. There may be backpressure from the Prefetcher in configurations where the number of Fetch Tags is lower than the number of Nodeslots.
3. **FETCH_NB_LIST**: while transitioning to this state, the NSB issues a request to the Prefetcher to fetch the list of incoming message addresses for the given node from the start address defined in the Nodeslot programming. The Nodeslot is held in this state until the NSB receives a done response from the Prefetcher, signalling that either 1. (OK) all incoming addresses are stored in the associated Fetch Tags or 2. (PARTIAL) the Fetch Tag FIFOs are full, and the Prefetcher will continue to fetch addresses as these are consumed by the NEIGHBOUR_FETCH stage.
4. **FETCH_NEIGHBOURS**: once the done response is received from the Prefetcher, the Nodeslot transitions to this state and the NSB issues another request to the Prefetcher, now to fetch the incoming messages at the addresses already stored in the Fetch Tag. The Nodeslot is held in this state until the NSB receives another done response from the Prefetcher, signalling that either 1. (OK) all incoming messages and edges have been stored in the associated Fetch Tags or 2. (PARTIAL) the Fetch Tag FIFOs are full, and the Prefetcher will continue to fetch incoming messages and edges once the data is consumed by the Aggregation Engine. If the last incoming

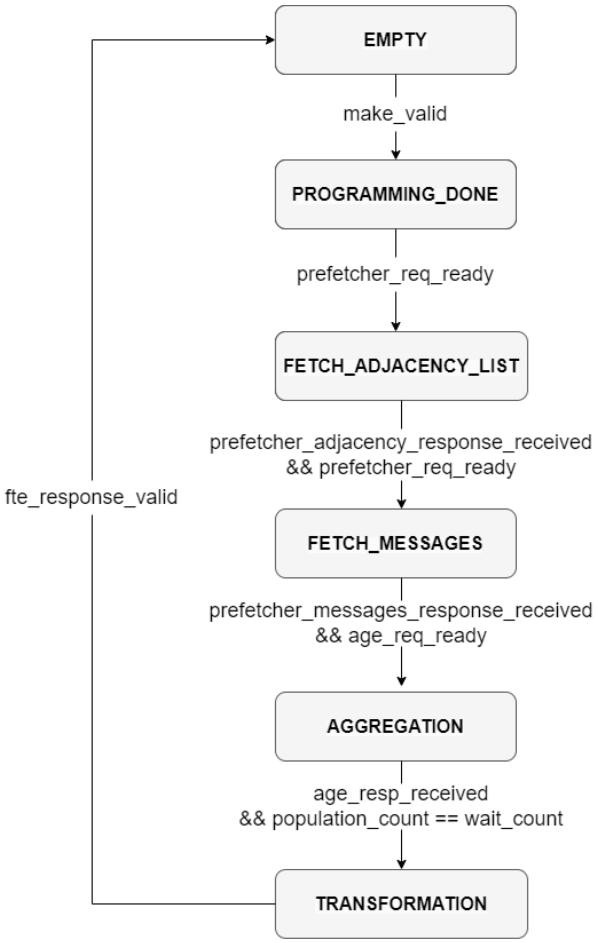

Figure 7: Node Slot state machine. Each box represents a state, with the transition conditions shown alongside the arrows.

message is fetched after the initial done response, the Prefetcher issues an additional status update to the Node Scoreboard so the **ALL_MESSAGES_FETCHED** mask can be updated in the Node Scoreboard for Debug purposes.

5. **AGGREGATION**: when transitioning into this state, the Nodeslot issues a request to the Aggregation Engine to begin performing the aggregation function over all incoming messages of the node stored in the Feature Prefetcher. The Nodeslot is held in this state until a done response is received from the Aggregation Engine, signalling that the aggregated feature embedding has been stored in the Aggregation Buffer.

6. **TRANSFORMATION**: with the aggregated feature embedding stored in the Aggregation Buffer, the Nodeslot transitions to this state when the transformation request is accepted by the Transformation Engine. The request is issued by the NSB for a combined group of Nodeslots when the number of available aggregations in the buffer matches the transformation wait count parameter. This state is held until a done response is received, signalling that the updated feature embedding has been stored in the Transformation Buffer.

3.2.3 Layer Configuration

In addition to controlling the Nodeslot state machines, the NSB drives the Prefetcher to fetch multi-precision weights at the start of each layer computation through a set of set-auto-clear registers. This flow requires the host to first set the **CTRL_WEIGHTS_FETCH_PRECISION** to the required precision, then write to the **CTRL_FETCH_WEIGHTS** registers, which is automatically cleared by the register bank logic. When this pulse is detected, the NSB triggers its internal state machine, asserting the **CTRL_WEIGHTS_FETCH_DONE** register when finished. The host is required to periodically read this register until it finds its value set to 1, at which point it writes to **CTRL_FETCH_DONE_ACK**, which is also auto-clearing. Finally, the hardware de-asserts the done register when the ACK signal is received.

3.2.4 Example Host Pseudocode

Algorithm 1 shows how the inference workload can be offloaded onto **AGILE** from the Host device. First, the Node Scoreboard is programmed with a number of global parameters. Layer configuration registers are programmed into the NSB before each layer computation. Each node is then programmed into the first available Nodeslot, according to a mask of free Nodeslots, which is updated asynchronously whenever a Nodeslot is programmed or freed.

Algorithm 1 Host programming pseudocode

```
Require: global parameter 𝒫, layers ℒ, graph 𝒢
  nsb_regbank.global_parameters ← 𝒫
  for layer in ℒ do
      nsb_regbank.layer_config ← layer
      prefetch_layer_weights()
      for node in 𝒢 do
          nodeslot_idx ← choose_nodeslot(free_nodeslots)
          nsb_regbank [ nodeslot_idx ] ← node_programming [ node ]
      end for
  end for
```

3.3 Large Graph Handling

3.3.1 Memory Requirements

The requirement to handle large graphs ($> 1M$ nodes) demands a reformulation of how feature and adjacency data is stored and communicated within the accelerator. Previous works have made extended use of on-chip memory. For example, GenGNN [10] contains a CSR table populated with a list of neighbours for each node. However, the memory capacity on high-end FPGAs such as the Ultrascale+ (shown in Table 3) is such that computation on large graphs becomes unfeasible.

Resource	Blocks available	Storage/block	Total Capacity
BRAM	2688	36 Kbits	12.39 MB
URAM	1280	288 Kbits	47.19 MB
DRAM	4	16 GB	64 GB

Table 3: Total on-chip and off-chip memory capacity on the Xilinx U250 board

It is worth comparing the resource capacity on the U250 board with the memory requirement for several large graphs, as shown in Table 4. It can be seen that the adjacency list fits within off-chip DRAM for all the datasets, although only the Flickr graph can be feasibly stored on-chip. The input feature data can also be fully stored on off-chip memory for all the datasets except ogbn-papers, due to its high node count ($> 100M$). It is worth noting that the reported figures show the memory requirement for only the input data to the GNN, however, the total memory requirement grows linearly with the number of GNN layers in the model since intermediate activations must be stored.

Graph	Nodes	Edges	Features	Feature memory [MB]	Adjacency memory [MB]
Flickr	89,250	899,756	500	170.2	3.6
Yelp	716,847	13,954,819	300	820.4	66.5
Reddit	232,965	113,615,892	602	534.9	487.6
AmazonProducts	1,569,960	264,339,468	200	1197.8	1,323.5
ogbn-products	2,449,029	61,859,140	100	934.2	324.5
ogbn-papers	111,059,956	1,615,685,872	TBC	57519	10,400.7

Table 4: Adjacency and input feature memory requirement for several large GNN datasets, assuming floating-point precision. The feature memory is evaluated from the node count and feature count. The adjacency memory represents the requirement for the list of edge tuples, with the minimum bit widths used to encode each node in the tuple according to the node count for the dataset.

In addition to on-chip RAM blocks and off-chip DRAM, **AGILE** can access Host RAM by a DMA Core that acts as an AXI/PCIe bridge. This extends the total available memory to any required capacity, at a latency cost trade-off.

3.3.2 Adjacency Data Encoding

AGILE requires access to the adjacency data of the graph to determine the memory address for incoming messages during the aggregation phase. A number of approaches were considered to handle the communication of graph data to the accelerator. The following approaches were considered:

1. **Edge tensor**: adjacency data is typically represented in datasets as a tensor of shape [2, EDGE_COUNT], which corresponds to a list of tuples where each linked node is encoded by a Node ID (the bit width for each node ID must be set to appropriately encode every node in the graph). In undirected graphs, the edge tensor is duplicated so that each node in each tuple is included as both a source and destination node. When sorted, this then corresponds to a list of all neighbours of

each node, encoded by Node ID. As shown in Table 4, the full edge tensor for large graphs can be feasibly stored in off-chip memory.

2. **Adjacency matrix**: graphs are often represented mathematically as an $N \times N$ binary matrix, with a 1 representing a link between the nodes associated with the corresponding row and column. Previous accelerators such as by Zhang et al. [22] have used graph partitioning mechanisms to store sections of the adjacency matrix on-chip. However, the incurred memory cost grows in $O(N^2)$ with the number of nodes, N. For example, the memory requirement for the Reddit graph is 6.7GB compared to 487MB in the edge tensor format. Additionally, the majority of the data is zero due to the sparse structure of graphs.
3. **Neighbourhood bit-mask**: each row of the adjacency matrix corresponds to a binary mask of all neighbouring nodes. This mask can be stored in the Node Scoreboard as part of the Nodeslot programming. Assuming support for up to 1M nodes, this incurs a requirement of 130kB of data per node or 8.3MB for all 64 Nodeslots.

Storage of the adjacency matrix or per-node neighbourhood bit mask was ruled out due to the high memory capacity requirements and poor bandwidth utilization. It was considered that the Host can program all neighbouring node IDs directly into the Node Scoreboard, however, it is not feasible to allocate enough on-chip RAM for the upper limit of 1M neighbours per node - this leads to a storage requirement of 167MB for all 64 Nodeslots. It would be possible to use a smaller storage element by allocating a lower neighbour limit, such as 1024, which leads to a memory requirement of 163kB. This would be enough to support most nodes, since the highest average node degree from the datasets in Table 4 is 488 (for the Reddit graph). However, the Nodeslot state machine would need to be updated so that once incoming messages were aggregated for all programmed nodes, the accelerator could request subsequent neighbours to be stored back into the Node Scoreboard. Since the previous neighbour IDs would be overwritten, the process would need to be repeated after feature transformation, to compute the outgoing messages.

To limit the requirement for high-latency communication between the Host and accelerator through PCIe, the chosen approach was that **AGILE** will fetch the adjacency list for each node from DRAM. For this purpose, the Host will program the start address for each Nodeslot's list of neighbouring node IDs as part of the Nodeslot programming.

An alternative approach was considered in which the accelerator does not require access to the graph adjacency data. By requiring the Host to program all incoming messages for each node in a contiguous range on off-chip memory, the Host can simply program each Nodeslot with a **INCOMING_MESSAGES_START_ADDRESS** field and the node degree (i.e. neighbour count). This leads to improved memory bandwidth since all incoming messages can be fetched from a single AXI transaction with the appropriate burst length. However, the memory requirement grows exponentially since the incoming messages for a given node will be stored once for each of its neighbours. The feature memory requirement for the AmazonProducts dataset grows from 1.2GB to 202GB assuming floating-point precision, for a single layer. It would not be feasible to perform this memory arrangement as a pre-processing step prior to inference on the graph, since a single layer is likely to saturate the memory capacity of Host RAM. To support this approach, the Host would be required to perform the memory manipulation in real time, prior to programming a node into one of the Nodeslots in the NSB. This would introduce a large increase in power consumption, as well as a major performance bottleneck if the memory arrangement cannot be pipelined while every Nodeslot is busy.

4 Microarchitecture Specification

The following micro-architectural specification details how the features and requirements discussed in Section 3 are achieved at the circuit level.

4.1 Overview of Dataflow

As previously discussed, the Node Scoreboard is responsible for allocating resources within the accelerator and driving internal interfaces with all other functional units to perform aggregation and transformation functions. Additionally, there are direct interfaces between the Prefetcher, AGE and FTE to bypass the NSB during data transfer. The main computational steps in a node's lifetime within the accelerator are listed below (i.e. after layer and global configuration registers have been programmed into the NSB).

1. **NSB** $\rightarrow$ **PREF**: Request to fetch adjacency list.
2. **NSB** $\rightarrow$ **PREF**: Request to fetch incoming messages using offsets stored in Adjacency Queue.
3. **NSB** $\rightarrow$ **AGE**: Request to aggregate features with given aggregation function.
4. **AGE** $\rightarrow$ **PREF**: Request for incoming messages stored in Message Queue.
5. **PREF** $\rightarrow$ **AGE**: Response with requested features through the Message Channel.
6. **AGE**: When aggregation is complete, place aggregation results into the next available slot in the Aggregation Buffer.
7. **AGE** $\rightarrow$ **NSB**: Response signalling that aggregation for each Nodeslot is complete.
8. **NSB** $\rightarrow$ **FTE**: Request transformation once the count of buffered aggregations reaches the configuration parameter.
9. **FTE** $\rightarrow$ **PREF**: Request layer weights stored in the Weight Bank.
10. **PREF** $\rightarrow$ **FTE**: Sequence layer weights in required order for systolic modules within Transformation Engine.
11. **FTE**: Write transformed features back into DRAM through the AXI Write Master interface or into the Transformation Buffer, according to the configuration parameter.
12. **FTE** $\rightarrow$ **NSB**: Response packet signalling that transformation is complete for each Nodeslot.

See Sections 3.2, 4.3, 4.4 and 4.5 for further details on the microarchitecture of each functional unit.

4.2 Internal Interconnects

AGILE contains two AXI interconnects, the **Memory Interconnect (MI)** and the **Register Bank Interconnect (RBI)**. The first has several masters and a single slave, connecting functional units of the design to the DRAM controller IP for memory access. This interconnect has 34-bit address width and a 512-bit data bus. RBI has a single master connected to several slaves, to provide access to the register banks within each functional unit of the design. In the simulation environment, the master is connected to the Testbench for stimulus driving. In the physical implementation, this is connected to the Host device through a PCIe/AXI-Lite bridge. The RBI follows the AXI-L protocol, which only supports a subset of AXI features required for register programming. The address and data busses are both 32 bits and burst transactions are not supported. In both interconnects equal priority was assigned to

all the masters. A Round-Robin arbitration strategy was chosen to ensure fair access in the event of multiple masters requesting access to the same address space.

4.3 Prefetcher

The Prefetcher is driven through a valid-ready request/response interface with the NSB. The main sub-units within the Prefetcher are its multi-precision Feature Banks and Weight Banks, responsible for fetching and storage of incoming messages and feature update weights, respectively. Three AXI read masters are instantiated within the Prefetcher, for Adjacency List, Messages and Weights fetching. The first two are driven by the Feature Banks (see Section 4.3.2), while the last is driven by the Weight Banks. An AXI-L interface is used to control the Prefetcher's internal register bank, containing layer configuration and control flags. Finally, the Prefetcher interfaces with the Aggregation Engine (AGE) and Transformation Engine (FTE) through its Message Channels and Weight Channels, respectively.

4.3.1 Weight Bank

Upon a request from the NSB, the Weight Bank fetches the matrix of weights required to run inference on a fully-connected layer. This takes place during the layer configuration phase, before Nodeslot programming. The weights are later used by the Feature Transformation Engine (FTE) to update each node's embeddings. Each row of the weight matrix is stored in a separate Ultraram FIFO, such that the weights can be flushed in the required order through the Weight Channel for consumption by the systolic arrays in the FTE.

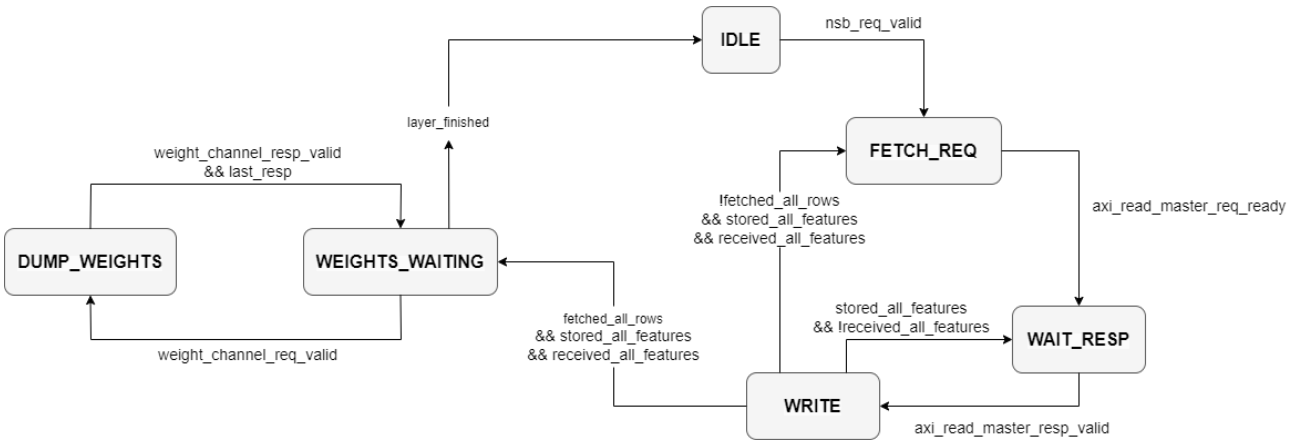

Figure 8: Weight Bank State Machine

As shown in Figure 8, the weight bank cycles between FETCH_REQ, WAIT_RESP and WRITE states while the weights are fetched. Each request to the AXI read master is for a single row in the weights matrix, i.e. up to 1024 features (or 4kB). Each AXI response beat contains up to 16 features, hence the required number of AXI beats is dynamically determined depending on the feature count. After receiving each response beat, the state machine transitions from the WAIT_RESP state to WRITE, where each of the 16 features is pushed into the FIFO over 16 cycles. After storing the last feature in the last expected response beat, the state machine either transitions back to FETCH_REQ (if there are more rows pending) or into WEIGHTS_WAITING. In the latter case, the weight bank waits for a request from the FTE to dump the weights over the Weight Channel.

In the DUMP_WEIGHTS state, the row FIFOs are pulsed such that weights arrive diagonally in the systolic modules in the FTE (see Section 4.5.1. This is achieved by an instance of the Systolic Module

Driver (see Section 4.7.4). During the weights dumping phase, the internal FIFO read pointers for each row are updated, but no data is overwritten in the Ultraram blocks. When transitioning back to the WEIGHTS_WAITING state, the pointers are reset to their original value. As such, the weights are immediately ready to be re-used for updating subsequent nodes in the same layer, without the requirement for repeated fetching.

4.3.2 Feature Bank

Before a node's incoming messages can be aggregated, the Prefetcher receives a request from the Node Scoreboard to fetch them from DRAM and store them in a local memory. There are 8 Fetch Tags in each Feature Bank, coupled directly to an Aggregation Manager in the AGE through the Message Channel. All Tags can issue requests concurrently, which are arbitrated through a Round-Robin Arbiter to provide access to the AXI Read Master. Once Aggregation Engine (AGE) starts requesting feature data, following from an aggregation request by the Node Scoreboard (NSB), these are issued sequentially from the Prefetcher by the corresponding Sequencer.

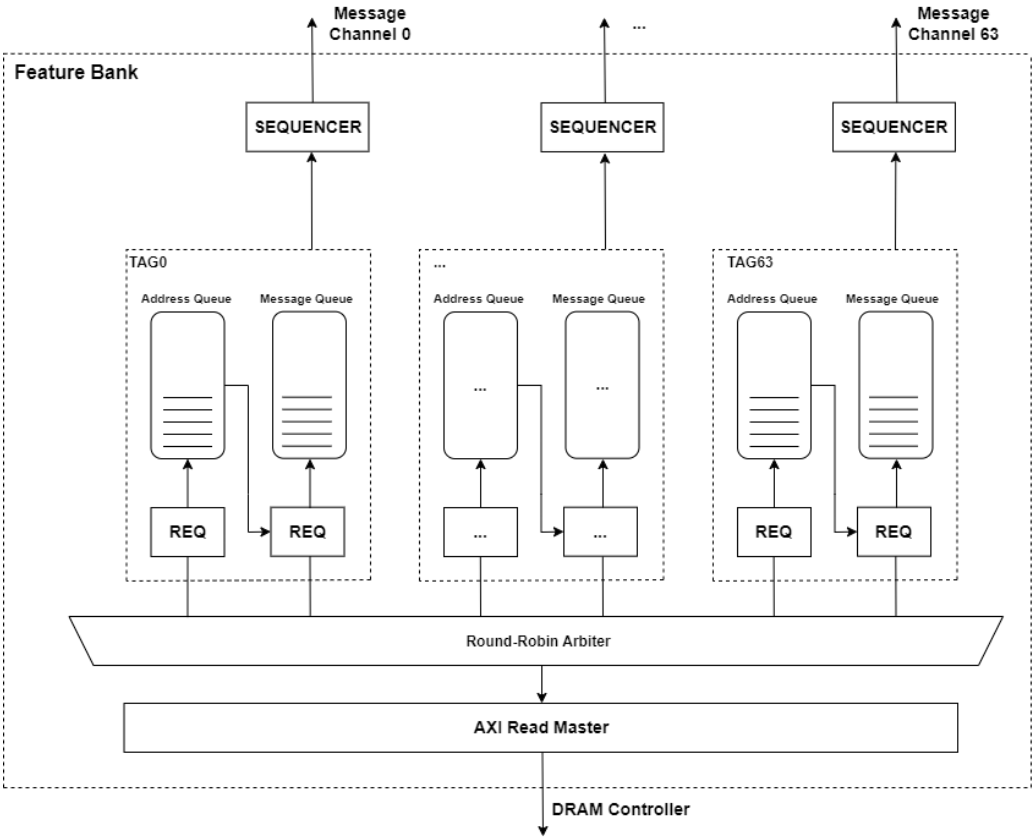

Figure 9: Prefetcher Microarchitecture for a configuration with 63 Fetch Tags. The AXI Read Master is driven from arbitration of the Request Engine, and the Sequencer reads data from the corresponding Fetch Tag according to the request from the Aggregation Engine.

Each Feature Bank receives requests through the Valid-Ready interface with the Node Scoreboard. First, the NSB requests the Prefetcher to fetch the adjacency list for the given node, while the Nodeslot is in FETCH_NB_LIST state. The payloads include the neighbour count and start address, which are transferred to the request logic for the Address Queue in the associated Fetch Tag. The request logic determines the number of bytes to be requested from DRAM from the starting address, given that

each adjacency pointer occupies 4 bytes. For example, if a Nodeslot has **NEIGHBOUR_COUNT** = 256 at **START_ADDRESS** = 0x080000000), the Address Queue will be populated with data in the range 0x80000000 $\rightarrow$ 0x80000400 (i.e. 1kB). The Prefetcher sends a done response for the adjacency list fetch request once either (1) the full list has been stored in the Address Queue, or (2) the Address Queue is full, and the remaining message addresses will be fetched once the Nodeslot transitions to FETCH_NB_LIST state and the incoming messages start being fetched and stored in the Message Queue.

After the done response for the adjacency list fetch is sent, the NSB will eventually issue a fetch request for the incoming messages when the Nodeslot transitions to the FETCH_NEIGHBOURS state. The Request logic for the Message Queue uses the addresses in the Address Queue to issue requests to the AXI Read Master, along with the incoming messages address programmed in the Prefetcher register bank. Similarly, a fetch done response is sent when either (1) all incoming messages have been stored in the Message Queue, or (2) the Message Queue is full, and subsequent messages will be fetched once the Aggregation Engine starts consuming the features. In case (2), a PARTIAL response is encoded in the status field to alert the NSB that subsequent status updates will be issued by the Prefetcher, signalling when the last incoming messages have been actually stored (this will take place when the Nodeslot is in the AGGR state).

4.3.3 Fetch Tag

The adjacency and message queues in each Fetch Tag were implemented using the **UltraRAM** blocks in the Ultrascale+ FPGA, such that incoming messages can be easily sequenced to the AGE. As shown in Table 3 (from Section 3.3 on Large Graph Handling), there are 1280 blocks available, each with a capacity of 288 Kbits, for a total capacity of 47.1MB. The use of these hardened blocks incurs a low cost to the usage of LUT elements due to the required increment/decrement counters and write/read pointers, however, the overall usage is dramatically reduced compared to using the LUT elements to implement the memory. The total size of the message queue is upper bounded by a corresponding feature size of 512, with 256 neighbours. This corresponds to 262kB per Tag, or 4MB across all fetch tags, which is well within the memory resource budget. This sizing supports the Pubmed graph without any requirement for partial streaming, which has 500 features with a maximum node degree of 171.

4.3.4 Multi-Precision Support

The primary advantage of **AGILE** over other GNN accelerators is its support for node-wise multi-precision computation. Within the Prefetcher, there is a parametrizable number of Weight Bank and Feature Bank instances responsible for each supported precision format.

As shown in Figure 10, supporting multiple precisions requires a two-stage arbitration process to access the Adjacency and Message AXI Read Masters. The first stage arbitrates among Fetch Tags within each precision block (i.e. Feature Bank), while the second stage arbitrates across precision blocks.

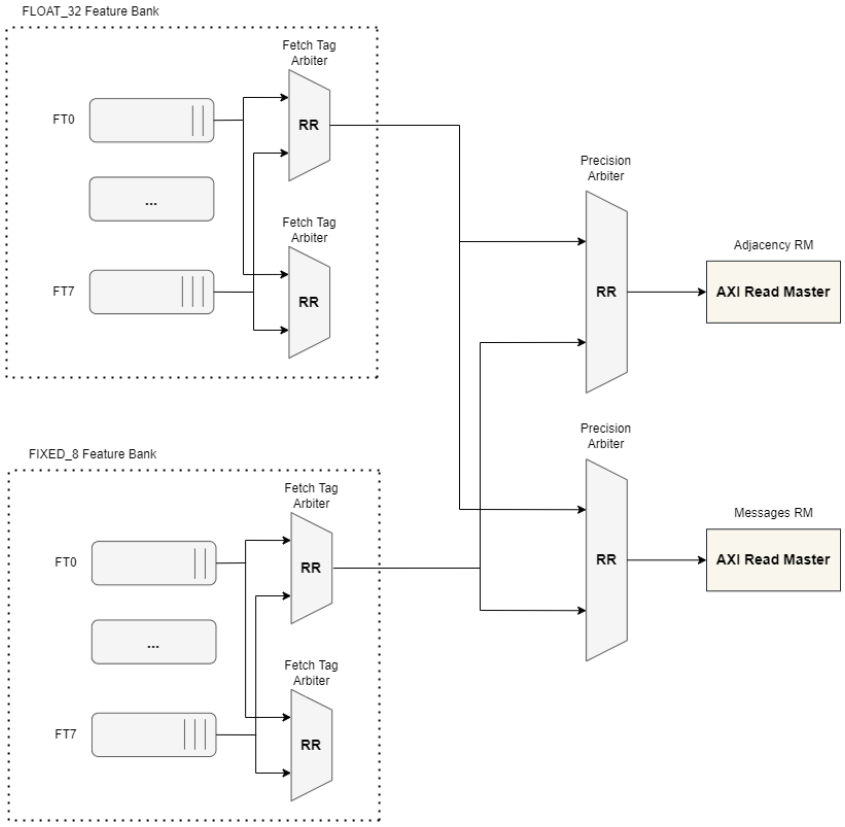

Figure 10: Two-stage arbitration process for access to the Adjacency and Message AXI Read Master from each of the Fetch Tags across all supported precision formats.

4.4 Aggregation Engine

The Aggregation Engine (AGE) is responsible for performing permutation-invariant aggregation functions over all the incoming messages of each node. The AGE receives requests over a direct interface to the Node Scoreboard (NSB) and requests data from the Prefetcher over the Message Channels (MC). Upon receiving a request from the NSB for aggregation of a given Nodeslot, the AGE allocates one of its Aggregation Managers (AGM) and a subset of its Aggregation Cores (AGC) according to the Nodeslot's numerical representation. After aggregation is complete, the AGE transfers the results to the Aggregation Buffer through one of its Buffer Managers.

The time taken to aggregate all incoming messages for a node in a graph is a function of the node's degree since more crowded nodes require a larger number of MAC operations. Previous accelerators have made use of static pipelines with a double buffering mechanism, where incoming messages are loaded into local memory while previously loaded messages are aggregated in a set of SIMD cores. In graphs with high variance in node degree, this leads to a large number of pipeline gaps since fast (isolated) nodes must wait for slow (crowded) nodes to release aggregation resources. To alleviate this, Aggregation Cores are asynchronously allocated by the AGE within the AGC Allocators (Section 4.4.2) following NSB requests.

4.4.1 Aggregation Mesh

One of the initial considerations in the design of the AGE was that the number of accumulators required for node feature aggregation is a function of the input feature count, which is a layer param-

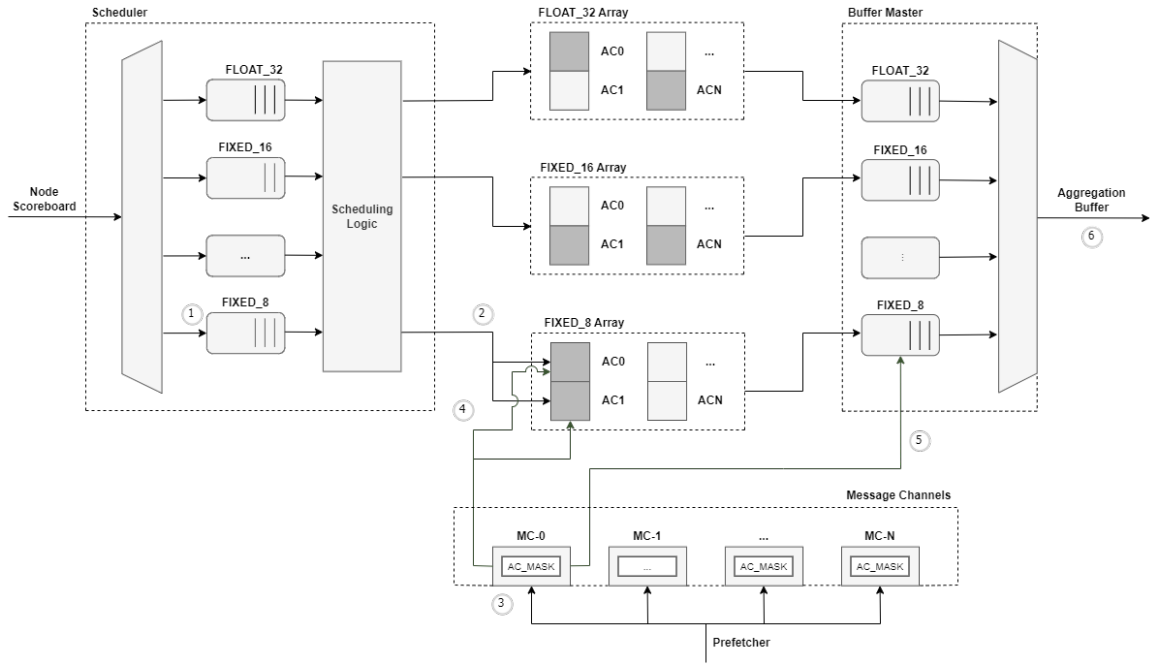

Figure 11: Microarchitectural diagram of the Aggregation Engine. NSB requests are sorted into Request Queues in the Scheduler according to Nodeslot precision prior to Aggregation Core allocation. The incoming messages are distributed into the ACs from the Message Channels.

eter. To achieve low inference latency, it was deemed crucial that the design should not require reprogramming of the FPGA between layer passes. Hence to achieve efficient aggregation with runtime-configurable feature counts, the AGE was designed to contain a network of Aggregation Cores (AGC) which are dynamically allocated at run-time (see Section for details). The AGCs are placed in a 2D mesh Network-on-Chip (NoC) topology, where nodes communicate via network packets. The AGE dynamically determines and allocates the required number of AGCs depending on the input feature count. This also enables aggregating nodes from multiple layers simultaneously, removing the requirement for unloading the accelerator after each layer pass.

In summary, the main advantages of the chosen architecture are as follows.

- **Avoid layer re-programming**: the required number of AGCs per node operation can be determined and dynamically allocated at run-time, depending on the layer's input feature count.
- **Asynchronous node aggregation**: while previous accelerators in literature have made use of static pipelines with a double buffering mechanism, leading to a large number of pipeline gaps due to the non-uniform distribution of node degrees, **AGILE** launches node aggregation as soon as enough resources are released, independently of other nodes on the accelerator.
- **Reduced congestion**: for each Aggregation and Buffer Manager to interface directly with each AGC requires complex logic that grows exponentially with the number of processing elements. By utilizing the discussed NoC architecture to handle communication between the PEs, the assumed resource growth is linear at the cost of marginally increased aggregation latency owing to packet propagation.

As shown in Figure 12, the lowest row in each Aggregation Mesh is occupied with Aggregation Managers, which are responsible for interfacing with Prefetcher Fetch Tags via the Message Channels and distributing features to the allocated set of AGCs. Additionally, the right-most column in occupied

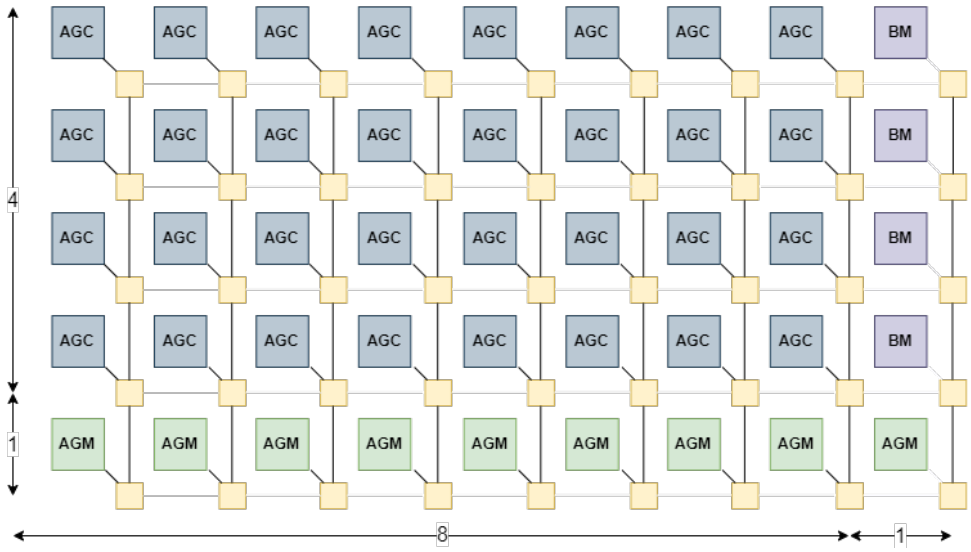

Figure 12: Aggregation Mesh, with routers shown in yellow, Aggregation Cores shown in blue and Buffer Managers shown in purple.

with Buffer Managers. Once an AGM is finished distributing features to AGCs, it will send a packet to each AGC instructing it to send its own features to the allocated Buffer Manager. The BM is then responsible for storing the received features in the Aggregation Buffer.

The AGE uses an open-source network router implementation provided by Galimberti, Testa, Zeni from Politecnico di Milano [23], which supports a Wormhole Switching architecture. Each packet is comprised of a Head flit, 0 or more Body flits, and a Tail flit. The head flit contains routing information such as source and destination node coordinates, while the tail flit may include data and/or "housekeeping" payloads to close the connection between two nodes. Each router is comprised of 5 ports (LOCAL, NORTH, EAST, SOUTH, WEST), each of which contains an input buffer which can only be allocated to a single packet. After receiving a head flit, the receiving port applies backpressure to all other ports, which must wait until the given packet is drained (i.e. the tail packet is forwarded). Galimberti, Testa, and Zeni's implementation additionally supports a parametrizable number of Virtual Channels to increase network throughput, however only a single VC was used in the Aggregation Mesh to reduce resource usage.

The routers in the Aggregation Mesh utilize the Dimension Order Routing algorithm, meaning that the destination port for each received flit is dynamically determined to first align the X coordinates of the incoming packet, then the Y coordinates.

4.4.2 Aggregation Core Allocation

The Aggregation Core Allocator sits at the frontend of the Aggregation Mesh, receiving NSB requests which are demultiplexed by the AGE according to precision. The Allocator has visibility of the mask of free AGCs constructed from the concatenation of their individual free flags. During the design, a race condition was identified between the Allocator and the AGCs, due to the time taken for allocation packets to propagate through the network. To overcome this, the Allocator keeps an internal mask register of allocatable cores, which is updated each time an AGC is allocated to a Nodeslot. This accounts for the case in which allocation for a new Nodeslot of the same precision is requested before the AGCs assert their allocation status flag. The allocatable cores mask is updated when a deallocation pulse is received by the Allocator, which takes place asynchronously to the allocation process, in the

event that the AGM sends its done response to the NSB.

After allocation, the Allocator issues an aggregation request to the required AGM, which encapsulates the original NSB request along with the coordinates of the allocated AGCs. These coordinates are contained in up to 64 allocation slots, since this is the maximum number of required AGCs for each Nodeslot, assuming a maximum feature count of 1024. An AGC count is also included in the request such that the AGM can consume the first N coordinate slots, where $N = \frac{\text{feature count}}{16}$.

AGC allocation within the Allocator takes place through a sequential round-robin mechanism. First, the required number of AGCs N is determined based on the layer feature count. Over the subsequent N cycles, the round-robin arbiter grants access to an available AGC, and the allocatable AGC mask is updated. Simultaneously, the X, and Y coordinates are determined according to the mesh dimensions set at compile time.

During implementation, an alternative approach was considered to extend the utilized Round-Robin Arbiter design to a multi-grant use case, such that all required AGCs could be allocated in a single cycle. This was left as future work to potentially reduce aggregation latency.

4.4.3 Aggregation Manager

The AGM's primary function is to drive the Message Channel interface with the Fetch Tags in the Prefetcher and transfer received features and scale factors as network packets to the AGCs. Messages are stored in the Fetch Tags at the granularity of AXI beats (512b), meaning 16 features are stored per Queue element, which is the same number of accumulators in each AGC. The number of flits required to send a 16-feature block was analyzed during design space exploration, since higher payload data widths lead to lower aggregation latency, at the cost of higher resource usage in the input port block buffers. After experimentation, the payload width was set to 64b, meaning 2 features are sent per flit, or 8 flits per AGC packet.

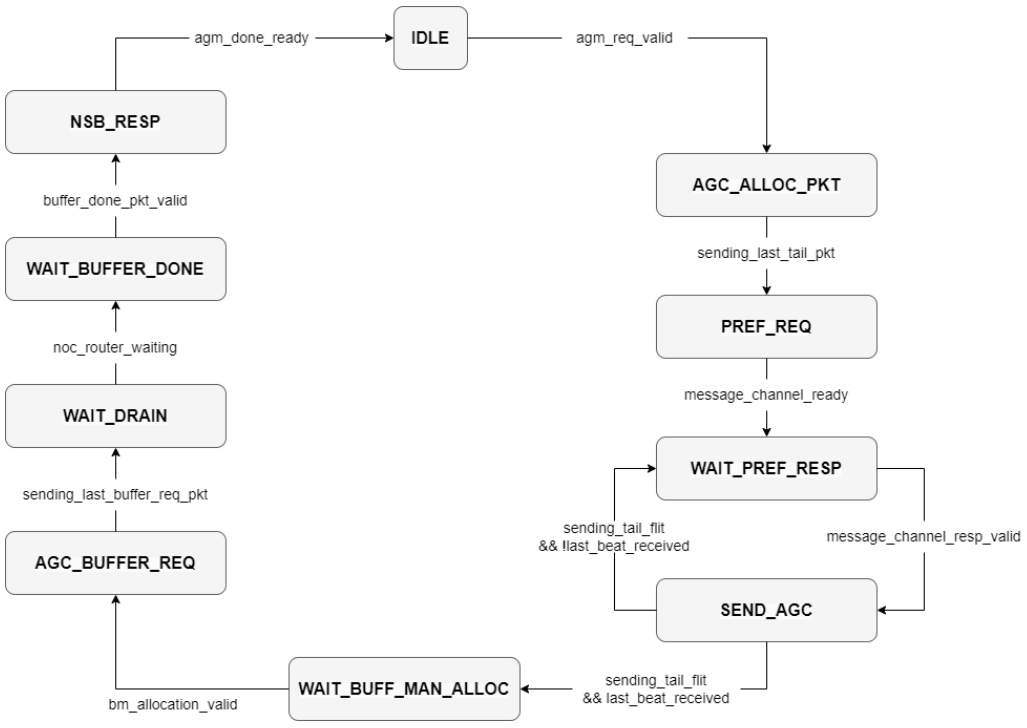

Figure 13: Aggregation Manager state machine

As shown in Figure 13, the AGM begins by sending a 2-flit allocation packet to each AGC granted by the AGC allocator, containing the Nodeslot and required aggregation function. After the Message Channel request is accepted by the Fetch Tag, the AGM cycles between the WAIT_PREF_RESP and SEND_AGC states. When transitioning to SEND_AGC, an auxiliary counter is triggered to ensure the correct features are selected from the message beat. The head flit in the feature packet contains the scale factor which is multiplied with every subsequent feature before aggregation. This is required for both GCN and GAT network architectures, where each feature vector is multiplied by degree and attention scalars respectively. An internal pointer is updated when transitioning away from the SEND_AGC state to ensure destination coordinates are selected from the correct allocation slot in the next packet.

When the Message Channel response has its **last** flag asserted, indicating the Message Queue is drained, the AGM transitions to the WAIT_BUFF_MAN_ALLOC state. This triggers the AGE to allocate a Buffer Manager allocation. When one becomes available, the AGM transitions to BUFF_MAN_ALLOC_PKT state, where a request packet is sent to each of the allocated AGCs containing the allocated BM coordinates. The AGM then remains idle for a period of time while the AGCs send aggregated features to the BM for buffering. When the AGM eventually receives a done packet from the BM, an NSB response is generated. Although the NSB response at the AGE interface is valid-only, backpressure is allowed at the AGM interface since the AGE needs to arbitrate among all AGMs sending response signals simultaneously.

Due to the Wormhole Switching mechanism of the NoC routers, the AGM is required to wait for the input buffer on the local port to be drained of any flits before being freed. This ensures no conflicts when the AGM is allocated to a new Nodeslot, and no latency is added in the typical use case since most flits are drained while the NSB response pulse is being arbitrated by the AGE.

4.4.4 Aggregation Core

Each Aggregation Core starts operating after receiving an allocation packet from an AGM, containing the allocated Nodeslot and aggregation function. The AGC contains 16 feature aggregators, which are updated with incoming features in the order they are received from the network. This relies on the assumption that order is preserved for flits within the same packet across the network, which is maintained for the described topology. An internal counter is updated after each incoming flit is received, which is used to drive the required feature aggregators in order. After receiving each feature, the AGC decodes the source node coordinate to reject any packets not originating from the allocated AGM. This acts to reduce computation error in the event of packet misdirection within the network.

As shown in Figure 14, when the last_packet flag is asserted in the head flit of any incoming feature packet, this signals that the AGM has finished draining the incoming messages in the Fetch Tag, so the AGC proceeds to the WAIT_BUFFER_REQ state until a buffering request packet is received. This then triggers an internal sent_flits counter as all aggregated features are transferred to the allocated Buffer Manager, at the chosen granularity of 2 features per flit. The AGC is freed as soon as the local port's input buffer is drained, without any further communication with the AGM required.

Each feature aggregator within the AGC contains a scale factor multiplier and a number of aggregator modules. **AGILE** supports user-defined aggregation functions, which can be defined using HLS or RTL modules and integrated within the feature aggregator wrapper module using a provided script. User-defined aggregators can be combinatorial or multi-cycle but must follow the specified interface where input features are driven with a valid-only protocol, i.e. no back-pressure is enabled. Required aggregators can be defined at compile time in a specified JSON file which is consumed by the build script. The base variant contains a sum aggregator, which can also support mean aggregation by subsequently pulsing a drive_division port in the final state of the AGC state machine. Finally, a

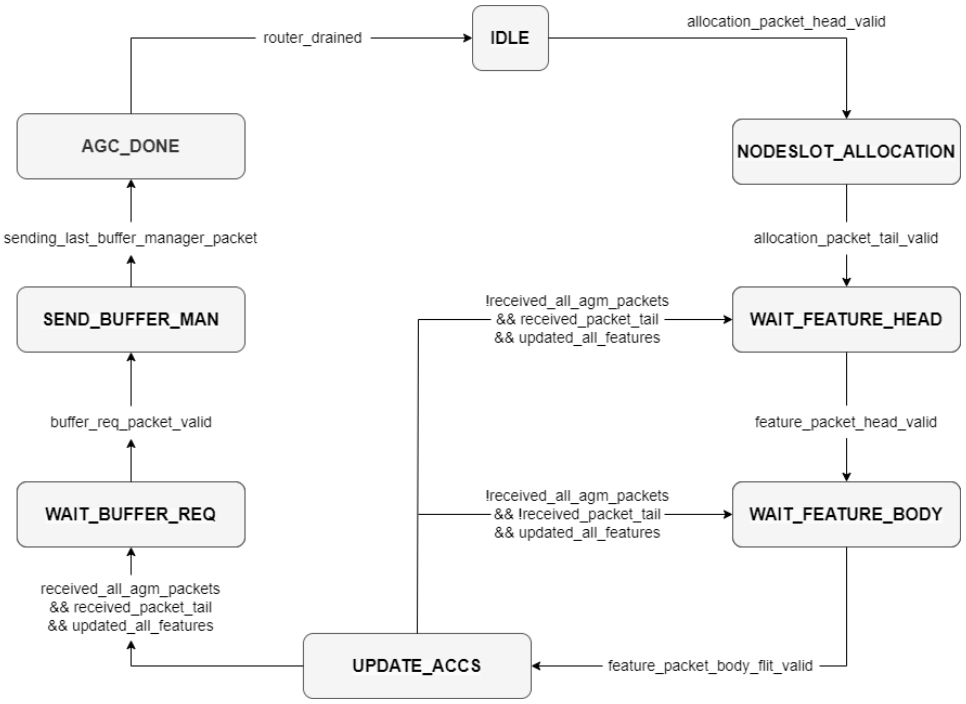

Figure 14: Aggregation Core state machine

"passthrough" aggregator is used during the first packet to reduce aggregation latency in case of multi-cycle aggregation functions.

4.4.5 Buffer Manager

After being allocated to a Nodeslot by the AGE, the Buffer Manager's primary role is to receive feature packets from associated AGCs and store these in the correct range within the Aggregation Buffer. Due to the topology of the network, feature packets are not necessarily received in the order in which they should be stored, since AGCs may be allocated in any mesh position depending on runtime state. As such, each Buffer Manager requires knowledge of the order in which AGC packets should be stored.

When the AGE couples a Buffer Manager to one of the AGMs, the allocation slot coordinates are transferred and registered into the BM. The BM then keeps a count of received flits from each AGC in the coordinate array. When addressing the buffer slot, the most significant address bits (i.e. AGC offset) correspond to the relative position of the AGC co-ordinates in the allocation slot list, while the least significant bits (i.e. feature offset) correspond to the received flit count for the incoming flit's source node.

As shown in Figure 15, the BM cycles between WAIT_FEATURES and WRITE states while packets are being received from the AGCs. The exit condition is given by an agc_done mask which is constructed as follows; first, a bitwise mask of valid allocation slots is constructed by subtracting 1 from the one-hot representation of the binary AGC count. Valid allocation slots are then considered done when their received flit counter matches the expectation, while the non-valid slots are tied to 1.

Each Buffer Manager is directly coupled to an Aggregation Buffer slot, hence after feature buffering is complete, the BM remains in WAIT_TRANSFORMATION state (i.e. allocated to its Nodeslot) until the feature count in the associated slot drops to 0. This indicates the FTE is done driving the aggregated features through the Systolic Array.

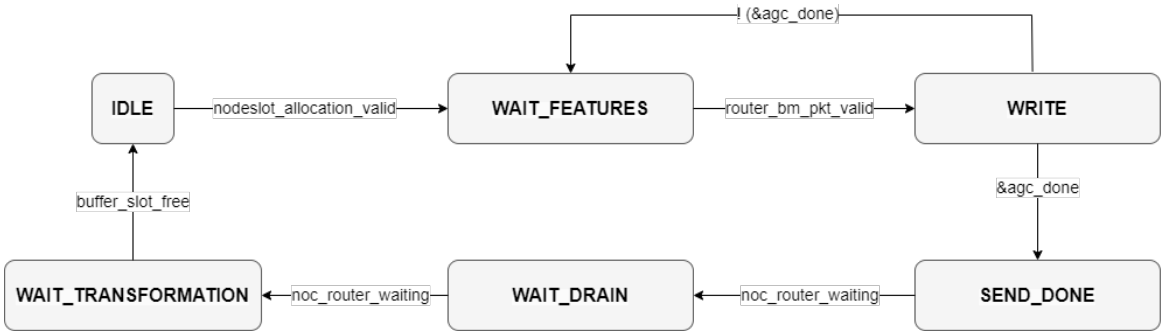

Figure 15: Buffer Manager state machine. A done response packet is sent to the associated AGM in SEND_DONE state after all features have been buffered. The BM is deallocated when the transformation is complete and the buffer slot is freed.

It should be noted that due to the Wormhole Switching mechanism, flits from different AGCs cannot be interleaved while reaching the BM since each Virtual Channel in the router can only be allocated to a single packet at a time. As such, a potential improvement to the BM would be to infer the feature offset for each AGC from a payload field in the head flit of each packet. This would remove the requirement for storing the coordinates of each allocation slot, reducing register usage by approximately 88 bytes per BM in the default configuration. This was left as potential future work.

4.4.6 Multi-Precision Support

The AGE instantiates a number of Aggregation Meshes, parameterizable at run-time, according to the number of supported precision formats. Each mesh is isolated from others, so packets cannot be transferred across precision boundaries. Aggregation requests from the NSB are routed to the appropriate mesh according to the node's precision. However, the arbitration for access to the NSB response interface assigns equal priority to all Aggregation Managers across all precision blocks.

4.5 Feature Transformation Engine (FTE)

The Feature Transformation Engine is responsible for fetching aggregated neighbour embeddings from the Aggregation Buffer and multiplying them with a matrix of learned parameters (received from the Weight Bank in the Prefetcher) to generate the updated feature embedding for each Nodeslot. The FTE starts working after a wakeup request from the Node Scoreboard (NSB), followed by an instruction to process the features present in the Aggregation Buffer. This request takes place when the number of available features matches the **NSB_CONFIG_TRANSFORMATION_WAIT_COUNT** parameter in the NSB configuration. After computation, the FTE stores the results in the Transformation Buffer and/or writes them back to memory through its AXI Master interface.

Fast matrix multiplication is achieved in the FTE through an array of Systolic Modules (see Section 4.7.3). The FTE supports the multi-precision targets described in Section 1 by dynamically allocating NSB requests to a number of Transformation Cores according to their precision. Each Transformation Core interfaces directly with its dedicated Aggregation Buffer. Finally, the transformation wait count parameter enables run-time control of the latency/power trade-off; lower latency is obtained at lower settings since nodes are immediately computed without waiting time, however, this requires propagation of the weights through the array a higher number of times.

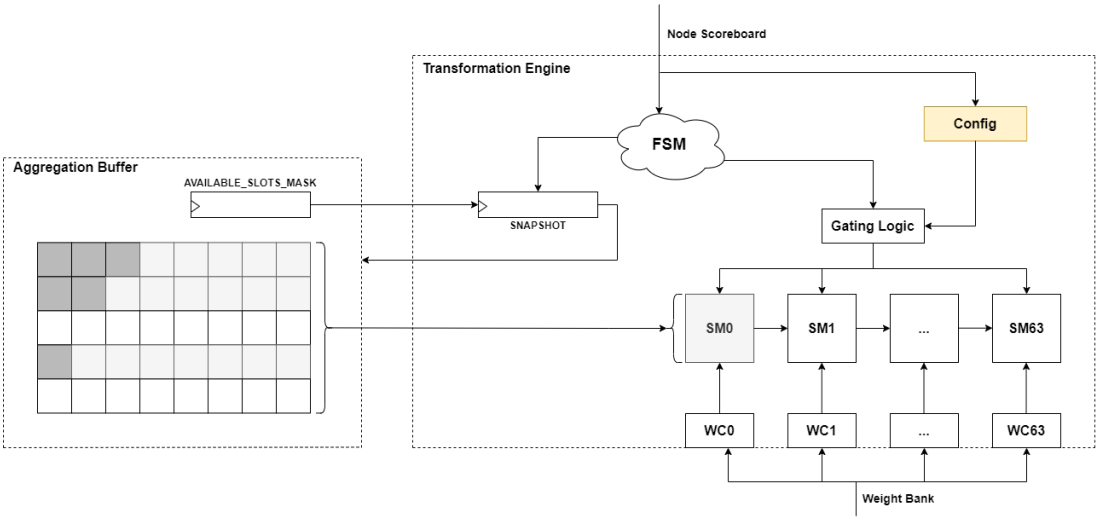

Figure 16: Microarchitecture of the Feature Transformation Engine. Following a request from the NSB, the FTE pulses the Aggregation Buffer FIFOs based on the captured snapshot. The Gating Logic shuts down non-required Systolic Modules (SMs) based on the output feature count. Weights are sequenced into the SMs from the corresponding Weight Channels (WC), which are coupled to the Prefetcher's Weight Bank.

4.5.1 Transformation Core

When the FTE is idle, and a request is received from the NSB, indicating the configured number of aggregated features are available, the mask of valid slots from the Aggregation Buffer is captured into a snapshot register. The snapshot is then used by the control logic to sequence feature data in the required pattern for multiplication in the systolic array. The snapshot mechanism is used so that, in the case when the **TRANSFORMATION_WAIT_COUNT** parameter is lower than the slot capacity in the Aggregation Buffer, the FTE can fetch aggregations from the correct offsets while the AGE continues to populate the buffer and change the contents of the available slots mask.

Matrix multiplication between weights and aggregated features is achieved in the FTE through up to 64 16×16 systolic modules. Weight data is received through the weight channels, each of which is linked to one of the Fetch Tags in the Prefetcher's Weight Bank. The weights are sequenced "from below" into each of the systolic modules at the required timing. The FTE instantiates a Systolic Module Driver (see Section 4.7.4) to pulse the Aggregation Buffer at the required timing to sequence features into the systolic module.

The dimensions of the Systolic Module are chosen based on the following considerations. As discussed previously, setting the **TRANSFORMATION_WAIT_COUNT** parameter to half the slot capacity of the Aggregation Buffer is equivalent to a Double Buffering arrangement between the A and FTE. Although the Host application can set this parameter low for lower latency on single-node computation, this will be set to 16 in the common use case. This reflects the assumption that at runtime, approximately $\frac{1}{4}$ of the 64 Nodeslots will be present in the Transformation Engine, with the remained spread across the Prefetch and Aggregation phases. Hence, the height of the systolic arrays is sized to 16 to support the common use case of double buffering between the A and FTE.

Equation 13 expresses the matrix dimensions of the multiplication, where $\hat{X}$ is the updated feature matrix, and W is the matrix of learned parameters. It can be seen the number of columns in the result matrix is determined by the number of output features for each GNN layer. Hence, the total number of columns of Processing Elements is determined by the maximum supported output feature count.

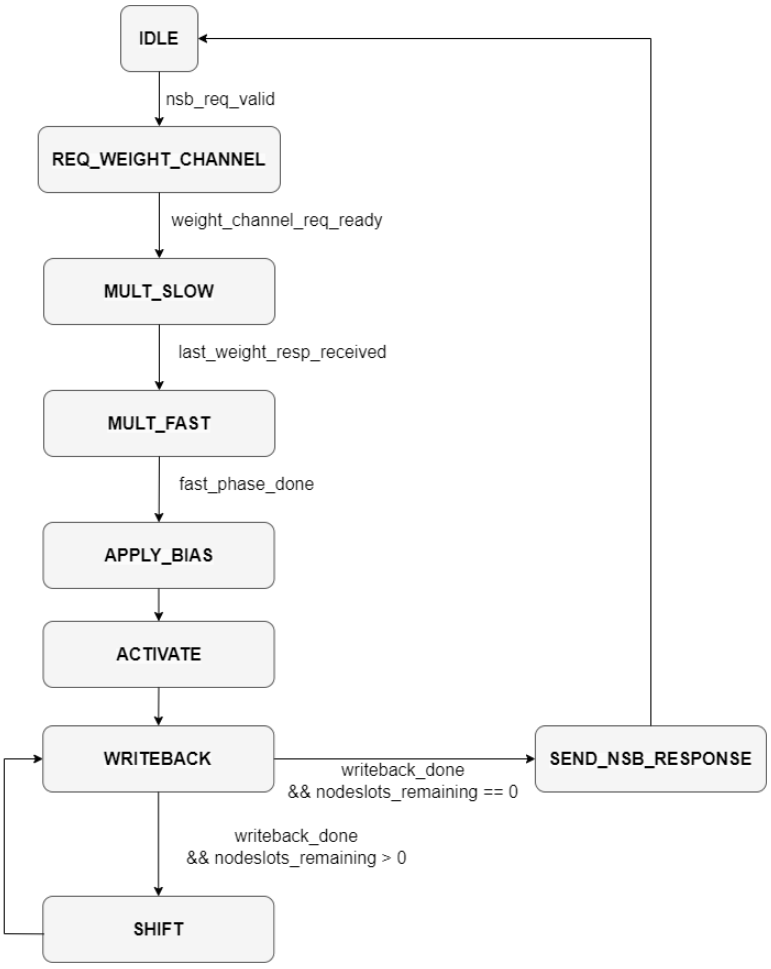

Figure 17: State machine of the feature transformation core

This is set to 1024, however, this is parametrizable at compile time.

$$\hat{X}W = (\text{WAIT\_COUNT} \times \text{IN\_FEATURES}) \times (\text{IN\_FEATURES} \times \text{OUT\_FEATURES}) \tag{13}$$

To support Message-Passing Networks such as GAT or GIN, the FTE may be required to store transformation results in a Buffer, to be used by downstream logic to compute outgoing messages. For simpler networks such as GCN, a node's outgoing message is simply its feature embedding, hence the FTE stores transformation results directly back to DDR memory. This leads to reduced computation latency as the node skips the non-required late stages of the pipeline.

The buffering and writeback logic in the FTE is achieved through a shifting mechanism. To support large feature counts, the FTE contains up to 16386 Processing Elements, meaning it becomes unfeasible to connect each row of PEs to any arbitrary buffer slot. After running synthesis on a fully connected architecture, it was found that the incurred multiplexers are too combinatorially deep to satisfy timing constraints at the desired frequency of 200MHz. This complexity was reduced by taking values from only the first row of PEs. After the first row is buffered, each row is shifted up using the overwrite mechanism in the MAC units (see Section 4.7.3).

4.6 Clock Domain Crossing at Register Bank Boundary

While running synthesis on the finished design, it was found that the register bank in the NSB could not meet timing constraints at the desired frequency of 200MHz, due to the combinatorially deep logic involved with multiplexing AXI transactions onto the large number of registers contained in the Nodeslot Scoreboard. As discussed in Section 6.1.2, this portion of the circuit is autogenerated by the AirHDL tool according to a JSON description of the required configuration registers. An initial attempt was made to reduce the complexity by reducing the number of Nodeslots to 32. This still did not meet timing, in addition to increasing latency, as fewer nodes could be pre-programmed while the accelerator was busy. The solution to the timing issue was to run the register bank circuit at a lower frequency of 50MHz while maintaining the remainder of the accelerator at 200MHz. This was achieved by introducing the multi-flop synchronization circuit shown in Figure 18 around each register in the register bank via an automation script that adds a wrapper around the autogenerated AirHDL code when building the register banks.

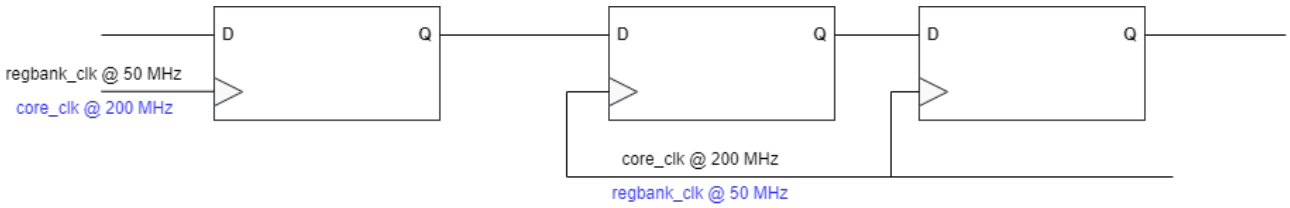

Figure 18: Double-flop synchronization circuit for data transfer across a clock boundary, comprised of two register stages clocked by the destination clock, with their data input tied to the data registered on the source clock.

The circuit in Figure 18 transfers data across the clock boundary with a reduced risk of metastability or data incoherency. Due to the difference in frequency, the setup constraints of the B register may be violated when data arrives, which would cause metastability as the register settles at the intended value, resulting in incorrect data being read by downstream logic. This risk is reduced by introducing the C register, since B is likely to have settled at the intended value by the time C registers its value, reducing the Mean Time Between Error (MTBE). Most registers in the NSB register bank are Read-Write from the software perspective, meaning data is transferred from the slow (50MHz) to the fast (200MHz) clock. Some registers are read-only, meaning data is written by the accelerator on the fast clock and read by software via the memory-mapped AXI-L Interface at the slow clock. In the latter case, the same circuit is utilized, although the source clock is at 20MHz (as shown in blue) while the destination clock is at 50MHz.

4.7 Library Components

During the implementation of the design, the need for a library of basic RTL components containing frequently-used functions was highlighted. In some cases, there was no readily-available Xilinx IP to achieve the desired functionality, while in others, the requirement for finer-grained control of timing and performance aspects called for custom implementation of the following base units.

4.7.1 AXI Read Master

The AXI read master is responsible to drive the AR (Read Address) and R (Read Data) channels in the AXI interface according to a user request. The upstream component defines the desired start address and byte count through a valid-ready request interface. The Read Master uses burst functionality to fetch the required data with the minimum number of AXI transactions. As such, the required beat

count is dynamically determined, and the AXI fields are driven appropriately. As defined by the AXI Protocol [24], a single transaction cannot cross a 4kB boundary, so in this event, the Read Master partitions the request into the required number of transactions before cycling through its internal state machine. Finally, the response beats are passed through to the downstream logic through a valid-ready response interface.

4.7.2 Hybrid Buffer

As discussed in Section 4.4, the Buffer Managers in the Aggregation Engine receive aggregated features in a non-deterministic order, due to the topology of the AGC mesh, however, the FTE consumes aggregated features in order. This requirement highlighted the need for a buffer implementation that can behave as an addressable RAM on the write interface but a FIFO on the read interface. This was achieved by the Hybrid Buffer, which contains a parametrizable number of buffer slots. Each buffer slot was implemented using BRAM blocks, to reduce LUT and Flip-Flop usage on the FPGA.

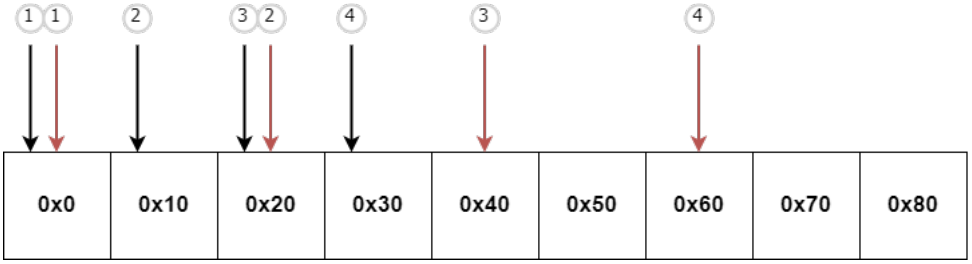

Figure 19: FIFO implementation using dual-port BRAM blocks. The red arrows show the write pointer following several push pulses, assuming a write width of 32B. The black arrows show the read pointer following several pop pulses, assuming a read width of 16B.

Additionally, dual-port BRAMs were used, enabling different widths and depths on the write/read interfaces. Since packet flits in the AGE mesh are 64-bit wide (i.e. containing 2 features each), these can be directly written into the Aggregation Buffer without the need for unpacking, reducing aggregation latency. These can then be read from the FTE at a 32-bit granularity by the Systolic Module Driver (see Section 4.7.4).

4.7.3 Systolic Module

The Systolic Array is a widely used architecture for evaluation of the core matrix multiplication shown in Equation 14.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} \tag{14}$$

This operation is achieved by the Systolic Module, which consists of a square grid of Processing Elements (PE). Each PE consists of a Multiply-Accumulate (MAC) unit, which takes input features from North and West PEs. Additionally, the incoming features are propagated "forward" and "downwards" to the East and South PEs, respectively. The matrix dimension n is parametrizable at compile time, as well as the arithmetic precision of the operands. See Figure 20 for an illustration of the required sequencing of matrix values, such that each PE contains the values of the resulting matrix after all features have propagated through the module.

The MAC units in each PE perform the multiply-accumulate operation over 2 cycles, using Xilinx floating-point adders and multipliers. A register is placed after the multiplication stage to meet timing

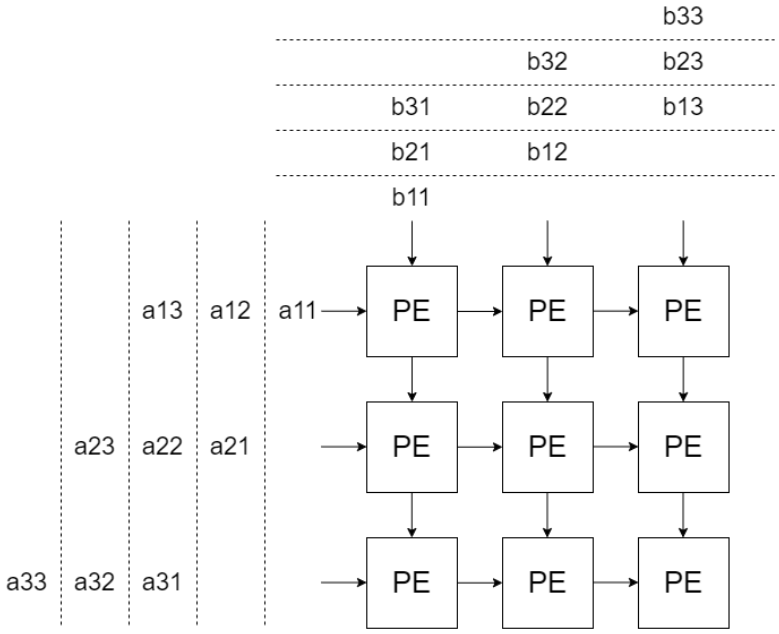

Figure 20: 4×4 systolic module. Each Processing Element takes two inputs, A and B. In each cycle, the inputs are propagated "forward" and "downward" to subsequent PEs. Additionally, A MAC core computes the product of A and B and adds the result to an accumulator register in the PE.

constraints, enabling the systolic module to be operated at 200MHz. In addition to the accumulators, each PE contains a bias adder and an activation core to fully perform the computation required for a Fully-Connected Neural Network layer. After driving the input matrix values, the upstream logic can pulse a bias_valid and then an activation_valid signal to overwrite the accumulator contents with the added bias and activated features, respectively. The activation core supports ReLU and LeakyReLU activations. Finally, the upstream logic can pulse a shift_valid signal to overwrite the accumulators with arbitrary data. This is used by the FTE during the writeback stage, as discussed in Section 4.5.

4.7.4 Systolic Module Driver

The Systolic Module Driver generates pulse signals in the format required to drive the read interface of the Hybrid Buffer such that data signals are made available with the required timing for the processing elements of a systolic module. This is achieved through a shift register of size HYBRID_BUFFER_SLOT_COUNT. After receiving a starting pulse, the least significant bit is set to 1. Subsequently, the register shifts after every shift pulse, up to a runtime-parametrizable pulse limit count parameter (this is set to the number of output features for the layer being executed). The driver should then pulse a subsequent HYBRID_BUFFER_SLOT_COUNT times until the register is flushed.

5 Hardware Verification

The verification of all functionality required by **AGILE** is based on two elements, (i) Formal Verification (FV) and (ii) constrained random stimulus.

The (i) **Formal Verification** targets involve writing design properties using the SystemVerilog Assertions (SVA) paradigm, to be processed by the Cadence JasperGold (JG) tool. JG attempts to generate formal proofs of the validity of these properties or, alternatively, counterexamples which can be used for debugging of functional issues. This flow is mainly used during the design bring-up stage to enable the identification of functional bugs early in the design process (prior to the development of the randomized testbench). Due to the high complexity and bandwidth requirements of the **AGILE** design, data coherency checks are not performed using FV, and these are left to the randomized testbench.

The (ii) **randomized testbench** contains a number of tests aimed at reaching high functional and code coverage within all units of **AGILE**. Stimulus is driven into the design using the AXI-L Verification IP, which is responsible for programming workload into the Nodeslots. The correct behaviour is verified by means of unit checkers. The checker has a handle to the input and output interfaces to each unit and generates expectation items based on the input transactions. The checker also generates observation items from the response transactions on the output interfaces. The expectation/observation scoreboard items are pushed into separate data structures, and any mismatches are evaluated.

From a stimulus perspective, several fields in the Nodeslot programming are randomized, including the requested numerical format, neighbour count and aggregation function. This is done to ensure correctness for all features individually and when taking place simultaneously within the accelerator. For example, sum aggregation should function correctly with or without other functions being computed. Additionally, the stimulus is further constrained along the way to account for corner cases.

Within the test hierarchy, a state machine is kept for each Nodeslot, which is updated according to the received programming through the AXI-L interconnect and the request/response loop between the NSB and all other functional units. Each unit checker generates expectation items based on the testbench view of the Nodeslot states. In the case of a mismatch between the RTL and testbench state machines for a Nodeslot due to an incorrect transition, this should result in errors to assist with the debugging.

5.1 Formal Properties

In the project, the primary target for Formal Verification (FV) was the library components (see Section 4.7), given their high rate of use and functional importance in the design, as well as smaller units within each of the primary sub-modules (Prefetcher, Aggregation Engine and Transformation Engine). FV tools were suitable for these use cases, given the low relative complexity and sequential depth. In each case, the main focus was on deadlock and liveness properties. Deadlock refers to the state in which units within the design are unable to proceed because of inter-dependency on each other to release resources or take some action. Liveness relates to the property of guaranteeing that the design reaches a certain desirable state.

- **AXI Read Master**: responsible for driving the AXI memory interface according to a request packet defining byte count and start address (see Section 4.7.1). An expectation of the required number of AXI transactions and beats per transaction is created based on the fetch request. Checks ensure that when a fetch response is issued back, the expectations match the real number of beats and transactions driven through the AXI channels.

- **Prefetcher Fetch Tag**: responsible for driving the request interface to the Adjacency and Message Read Masters (see Section 4.3.3). Checks ensure that (1) the requested number of bytes are consistent with the neighbour count and precision when a normal (adjacency or messages) response is issued, (2) partial responses are only issued when the requested byte count exceeds the queue capacity, (3) message fetch requests should not be issued until after the adjacency NSB response has been sent.

5.2 Node Scoreboard Checker

The following checks are in place in the NSB checker.

- **INVALID_REQUEST_STATE**: Requests to the PREF, AGE, Transformation Engine etc. occur at the correct state for each Nodeslot. For example, a request to the Transformation Engine should not be issued from the NSB before the Aggregation Engine request has received a response, causing the Nodeslot to transition to the TRANSFORMATION state.
- **<PREF/AGGR/TRANS>_REQ_PAYLOAD_MISMATCH**: The neighbour counts, start address and other payloads provided in requests to the Prefetcher, AGE and Transformation Engines are consistent with the Nodeslot programming from the AXI-L register writes and other events in the Nodeslot state machine.
- **INVALID_<TRANS>_REQ**: Requests to the Transformation Engine should not be issued until the required population count is reached in the Aggregation buffer. This is configured according to the NSB_CONFIG _POPULATION_COUNT parameter in the NSB.

5.3 Prefetcher Checker

The Prefetcher interacts with all other functional units in the design, as well as the AXI interconnect for Memory requests. The Prefetcher checker maintains checks for the following functionality:

- **CORRECT_AXI_REQ_COUNT**: Following from an NSB request, the appropriate number of AXI transactions are issued, according to the neighbour count and start address in the request.
- **NSB_RESP_PAYLOADS_MISMATCH**: In a response with **STATUS** = OK, the **fetched_neighbours** count is the same as the neighbour count in the request phase. **PARTIAL** responses only occur for Nodeslots in which the neighbour count is greater than the Fetch Tag capacity. Furthermore, if a PARTIAL response is received, the **fetched_neighbours** count should be saturated at this capacity, which signals to the NSB that the Fetch Tag is full, and the Prefetcher will continue fetching incoming messages once the AGE begins its computation.
- **INVALID_PARTIAL_UPDATE**: in cases when a Nodeslot's neighbour count is higher than the Fetch Tag capacity, the Prefetcher will issue a secondary update transaction after the initial done response to signal to the NSB that the full neighbour count has been reached. This update transaction should only take place for Nodeslots that received PARTIAL responses in the first place.

5.4 Aggregation Engine Checker

The following checks will be in place in the AGE Checker. Most importantly, this checker is responsible for data coherency checks on the aggregation results.

- **INVALID_PREF_REQ_PAYLOADS**: payloads in the AGE's request to the Prefetcher should match those received from the NSB.

- **INVALID_BUFFER_TRANSACTION_PAYLOADS**: after completing aggregation, the AGE should store the results in one of the Aggregation Buffers, according to the Nodeslot precision. The chosen precision should match the Nodeslot programming.
- **AGGREGATED_MESSAGES_MISMATCH**: the resulting aggregation of incoming messages for a Nodeslot, stored in the Aggregation Buffer, should match the internal expectation of the Testbench.

5.5 Feature Transformation Engine Checker

- **INVALID_BUFFER_PULSE**: the FTE should only fetch features from the correct Aggregation Buffer, according to the precision in the NSB request. Additionally, only buffers slots associated with the Nodeslots in the transformation request should be popped.
- **INVALID_WEIGHT_CHANNEL_REQ**: a Weight Channel request should only be issued for the precision in the latest NSB request.
- **INVALID_TRANSFORMATION_BUFFER_WRITE**: the FTE can optionally write transformation results onto the Transformation Buffers for further processing or directly back into DRAM, according to programming in its register bank. In WRITEBACK-only mode, no buffer transactions should be observed.
- **DATA_CONSISTENCY_**: the transformation results stored in the buffer or written out to DRAM should match the expectation of the testbench.

6 Implementation Milestones

The Design and Verification workload for the implementation of **AGILE** was divided into 5 milestones, structuring the project into manageable checkpoints such that functionality is built progressively. In each milestone, design tasks (i.e. RTL implementation) are preceded by allocated time for Microarchitecture specification, in which each unit is defined at the circuit level to meet the Architectural Specifications in Section 3. Verification work for each milestone was planned to take place simultaneously with Microarchitectural specification for the subsequent milestone. Implementation began on December 19th 2022, following a 6-week literature review period after project allocation on November 1st. The results from this literature review are reflected in Section 2.

6.1 Milestone 1: Infrastructure

The first milestone was dedicated to setting up the infrastructure that will be used for the remainder of the project. This includes all required Xilinx IP blocks and Register Banks to enable programming of the accelerator from the Host.

6.1.1 Xilinx IP

AGILE makes use of a number of blocks of IP from Xilinx, within both the design and Testbench. The following were instantiated using the IP Catalog in Vivado and declared within the RTL code.

- **DRAM Controller**: The DDR4 Controller IP provides access to DRAM memory through an AXI interface. The User Logic can issue AXI write or read transactions, making full use of burst functionality to maximise bandwidth. The controller performs a series of bandwidth optimisation steps and drives the fields on the DDR4 interface to access the appropriate bank groups.
- **DRAM model**: Xilinx provides a DRAM simulation model to emulate the latency incurred by accesses to off-chip memory. This can be configured with the same DDR4 parameters as the chip present on the required FPGA board (U250).
- **AXI and AXI-Lite Interconnect**: an AXI interconnect connects one or more memory-mapped AXI masters to one or more AXI slaves. Each slave is assigned an address range such that transactions from each of the masters are routed to the appropriate endpoint. Priorities can be assigned to each of the masters to resolve conflicts with multiple masters requesting to access the same slave at the same time.
- **AXI Verification IP (VIP)**: The AXI VIP is a verification component configured in either Master, Slave or Passthrough mode. The VIP is instantiated within the Testbench hierarchy and passed a reference to an AXI interface in the design. Within the stimulus stack, a VIP Agent can then be instantiated which contains functions that can be used to drive randomized AXI transactions. The VIP also acts as a protocol checker on the signals received from the slave.

It should be noted that the DRAM Controller was included in the design during synthesis for resource usage estimation however not during simulation. Due to issues with memory initialization with the DRAM model provided by Xilinx, it was chosen to use an AXI RAM model provided by Alex Forencich [25]. To emulate the random-access behaviour of DRAM, a wrapper was instantiated around the RAM model to add random latency between request and response transactions.

6.1.2 Register Bank Flow

A register bank is a group of memory-mapped registers within a hardware block used for configuration of global parameters, receiving instructions from the Host device and providing visibility of the hardware state for Debug purposes. Within the FPGA, these registers are implemented using Flip Flops within the Logic Elements.

All register banks for this project were implemented using Airhdl, a web-based service for register management. The tool autogenerates the SystemVerilog or VHDL implementation of register maps with an AXI-Lite interface from a user definition of each register field. Register maps can be defined either on the User Interface in the website or using JSON files. A number of Design Rule Checks (DRCs) are performed to ensure address space coherency, valid register names, etc. In addition to the SystemVerilog/VHDL implementation, the tool can also be used to generate HTML/Markdown documentation and C++ Header files to be used in the Host software stack.

Milestone 1 required the implementation of an automation script to integrate Airhdl with the Vivado toolflow. After register fields are manually updated on the Web Interface, the issues HTTP requests to autogenerate the SystemVerilog payloads, download the output files and update these within the Vivado project hierarchy. This enables iterating faster on the design when feature updates are required. Later in the project, this script was extended to create a Clock Domain Crossing wrapper around the auto-generated RTL, as discussed in Section 4.6.

6.1.3 Verification

At this stage, a Top-Level testbench was instantiated, which includes the Design Under Test (DUT), DRAM model and AXI-L VIP to drive Nodeslot programming. At this early stage, the DUT comprises of template RTL with dummy register fields declared in the register banks for each functional unit. The following test was implemented to verify the functionality associated with Milestone 1.

- **REGISTER_BANK_TEST**: write a sequence of randomized data into each register in every register bank within the design. After each write, perform fencing reads to verify data coherency.

6.2 Milestone 2: Matrix Arithmetic Kernel

The aim for Milestone 2 is to implement the core control and data paths within the accelerator by supporting the simple matrix kernel $(A + B)W$, where A, B, W are arbitrary $n \times n$ matrices with W representing the matrix of layer weights. This is done by formulating the matrix as a graph, where the number of matrices in the addition stage is equivalent to the neighbour count of each node, the number of rows is equivalent to the number of nodes, and the number of columns is equivalent to the feature count.

6.2.1 Design

Although this milestone supports a small subset of the functionality, it lays the foundation with the following functional units in place.

- **Fetch Tags** in the Prefetcher Feature Bank, coupled to **Aggregation Managers** in the AGE.
- Prefetcher **Weight Bank**.
- **Aggregation Cores**, **Aggregation Managers** and **Buffer Managers** interconnected through the **Network-on-Chip** in the AGE.

- **Aggregation and Transformation Buffers**.
- **Systolic Array** in the Feature Transformation Engine.

A number of simplifications were made at this stage. In the Fetch Tags, streaming messages with partial NSB responses is not yet supported (i.e. Nodeslots do not have a neighbour count higher than 64, which would saturate the Adjacency and Message queues). In the AGE, Aggregation Core allocation only supports the single-core case, i.e. Nodeslots can only have a feature count up to 16. Finally, Multi-layer and multi-precision support is left for Milestone 3 (section 6.3.

6.2.2 Verification

Within the top-level testbench, a new **MATRIX_KERNEL** test was instantiated to randomize the matrices and drive the stimulus in the compatible format required by **AGILE**. The test starts with the following configuration stimulus. Firstly, the <INPUT/OUTPUT> _FEATURE_COUNT is set to the required matrix column size and the start address for the weights fetch is written. Subsequently, the required number of Nodeslots are programmed, each with two neighbouring nodes and floating-point precision. Each incoming message from the neighbours to the Nodeslots corresponds to a row in the A and B matrices. The **AGGREGATION_FUNCTION** parameter is set to **SUM** for all Nodeslots. The **TRANSFORMATION_WAIT_COUNT** parameter is set such that multiplication begins once all messages are stored in the Aggregation buffer.

6.3 Milestone 3: Multi-Precision Aggregation and Transformation

6.3.1 Design

In this milestone, full support for multi-precision aggregation and transformation will be added to **AGILE**. In the Aggregation Engine, a larger pool of Aggregation Cores (AGCs) will be in place with varied numerical representations. MS3 requires the implementation of the Scheduler, which will sort NSB requests among the Request Queues and perform allocation over the AGCs. Additionally, a higher number of Message Channels will be in place, which should distribute messages across the Aggregation Cores. Finally, each Aggregation Core will support additional aggregation functions as defined in Section 4.4.

6.3.2 Verification

As previously mentioned, node aggregation has very variable latency due to the high variance in node degrees within a graph. This means that the request/response loop between the NSB and the A is not order-preserving. In the testbench, an associative array data structure is used to keep track of Nodeslots with ongoing work in the A. Response ordering can be enforced in the Transformation Engine checker since the order in which updated feature embeddings are stored in the Transformation Engine is deterministic. This is a function of the row order in which aggregated features are stored in the Aggregation Buffer due to the way that MLP weights are propagated through the Systolic Modules.

7 Results Evaluation

In this section, the evaluation set-up is described, and the benefits of the **AGILE** architecture are shown by comparing inference performance against CPU and GPU counterparts. Finally, a number of optimization strategies are described, which were undertaken to improve timing performance.

Several design variants were considered during the evaluation of the accelerator. The main degrees of freedom were in the number of aggregation and transformation channels for each precision block; the former is equivalent to the number of Fetch Tags in the Feature Bank and columns in each Aggregation Mesh, while the latter corresponds to the number of rows in the mesh and slots in the Aggregation buffer. A process of design space exploration was carried out with the aim of maximising the parallelization level while minimizing the resource usage on the U250 board.

7.1 Experimental Setup

Firstly, inference times were obtained on CPU and GPU devices by encapsulating GCN models in a Pytorch Lightning module, which abstracts boilerplate algorithms for training and validation. The model was trained for classification on the PubMed, CiteSeer and Cora datasets [26]. The cross-entropy loss function was used along with the Adam optimizer, with a learning rate of 0.005. Importantly, a custom prediction step function was implemented in the Lightning module for the estimation of inference timing. Prior to each forward pass, CUDA's **empty_cache** function is called to clear the cached memory in the GPU. Thus, the resulting times include the latency of off-chip memory access for incoming features and learned weights. Note that GPU warm-up time was not included, meaning inference times are taken after driver initialization is complete.

Inference latency on **AGILE** was obtained for several graphs from Modelsim simulation results at a frequency of 200MHz. This is equivalent to the maximum frequency obtained from Vivado's post-synthesis timing summary, suggesting the obtained results are representative of latencies post Place & Route. Note that prior to the forward pass on AGILE, a precision mask was stochastically applied with uniform distribution, meaning nodes were approximately evenly distributed across the precision channels.

7.2 Timing Performance

Table 5 shows the obtained latencies for inference on a single GCN layer with 64 input and output features. Note that CPU and GPU times vary in each iteration due to background processes and other running workloads; hence mean and standard deviation statistics are obtained over 100 trials. **AGILE** timings were obtained from simulation using a latency-optimized variant with 16 aggregation and transformation channels, with two precision blocks (floating-point and 8-bit integer).

	CPU (Intel Xeon @2.3GHz)	GPU (Tesla T4)	AGILE		
	Mean Latency ($\pm$ std) [ms]	Mean Latency ($\pm$ std) [ms]	Latency [ms]	Speedup (CPU)	Speedup (GPU)
KarateClub	4.00 $\pm$ 4.26	3.28 $\pm$ 0.98	0.0826	48.43$\times$	39.71$\times$
Pubmed	48.50 $\pm$ 7.11	3.34 $\pm$ 1.31	7.35	6.60$\times$	0$\times$
Cora	4.80$\pm$ 1.60	2.28 $\pm$ 0.69	1.11	4.32$\times$	2.05 $\times$
Citeseer	5.30$\pm$ 2.50	2.78 $\pm$ 1.32	0.963	5.50$\times$	2.89$\times$

Table 5: Inference latency for single-layer GCN model, with first-order statistics obtained over 100 iterations. The utilized **AGILE** variant contains 16 aggregation and transformation channels, with fixed-point and 8-bit precision.

It can be observed that GPU times are approximately constant across all graphs despite the wide range

of node counts. This suggests a large proportion of time is consumed for initialization operations which have $O(1)$ time complexity with respect to input graph size. This explains why the most significant speed-up (48.4x and 39.7x for CPU and GPU, respectively) was obtained with the KarateClub graph (34 nodes). Significant speed-ups were also observed for Cora and Citeseer, with $2,708$ and $3,327$ nodes, respectively. Latency for the Pubmed graph (19,717 nodes) was higher than on the GPU by approximately 4ms, however, 41.2 ms lower than the CPU baseline. See Section 8.1 for proposed optimizations left as future work, expected to bring computation latency closer to the GPU baseline.

7.3 Resource Usage

Table 6 shows the resource usage summary for an evaluated variant with 16 aggregation channels and 4 transformation channels per precision block at several supported arithmetic precisions.

Resource	**LUT**		**FF**		**BRAM**		**URAM**		**DSP**		**BUFG**	
Unit	**/1,728,000**	**[%]**	**/3,456,000**	**[%]**	**/2,688**	**[%]**	**/1,280**	**[%]**	**/12,288**	**[%]**	**/1,344**	**[%]**
FLOAT	1,347,285	78.0	1,852,681	53.6	896	33.3	1,168	91.3	3,076	25.0	2	0.2
FLOAT + FIXED_16	1,148,779	66.5	1,362,854	39.43	528	19.6	1024	80.0	1027	8.36	2	0.2
FLOAT + FIXED_8	1,299,369	75.2	1,448,142	41,9	621	23.1	1059	82.7	4	0.03	2	0.2
FLOAT + FIXED_4	1,105,890	64.0	1,249,340	39.1	528	19.6	1024	80.0	3	0.02	2	0.2

Table 6: Resource usage summary for floating point and mixed precision variants. LUT: Look-Up Table, FF: Flip-FLop, BRAM: Block RAM, URAM: UltraRAM, DSP: Digital Signal Processing units, BUFG: clock buffer

Usage of all resources is approximately equal for the 4b and 8b fixed-point variants, although the former had LUT usage reduced by 11.2% owing to its lower precision representation. Interestingly, LUT usage for the 16b variant was 8.7% lower compared to 8b, despite the higher precision obtained. This is explained by the fact that the Vivado synthesis flow casts arithmetic to DSP units at 9b precision and above. As expected, DSP usage was highest for the float-only variant.

Although the absolute memory requirements for all variants are equal, distinctions are observed in the resource usage due to non-deterministic optimizations made by the Vivado Synthesis tool. The synthesis report shows that in the floating-point variant, the adjacency queue and fixed-point weight bank were mapped to DRAM, while the message queue and floating-point weight bank were mapped to URAM. Meanwhile, the fixed-point weight bank was mapped to URAM in the 16b variant, with the floating-point weight bank implemented as BRAM.

7.4 Performance Optimizations

During performance evaluation, a characterization process was undertaken to determine the computational phases that consume the most latency on the device. On average, the most time-consuming phase was found to be Aggregation (50.3 % of Nodeslot lifetime), followed by Transformation (22.4%) and Nodeslot programming (18.8%).

Aggregation latency was reduced by altering the payload width per flit in the aggregation mesh. As discussed in Section 4.4.1, flit granularity is chosen as a trade-off between aggregation latency and resource usage. For larger flit sizes, higher congestion is expected due to the increased width of the data bus between router ports. Additionally, Flip-Flop consumption varies due to changes in the required input buffer size per port. While resource usage results in Subsection 7.3 were evaluated for the resource-optimized case (64b flit width), timing results in Subsection 7.2 were latency optimized at 512b flit width. This incurs an added storage requirement of 280 bytes per router, or 71.6kB for the evaluated variant with 16 aggregation and transformation channels. As discussed in Subsection 8.1, this requirement will be cast to BRAM resources for LUT/FF optimization in future work.

Although matrix multiplication in the FTE's Systolic Array is a fixed-latency operation, according to the required output feature count and the number of transformation channels, the number of transformation passes per model layer is parametrizable via the transformation_count_register in the NSB. As discussed in Section 4.5, this choice is a trade-off of latency, throughput and power consumption. While a reasonable choice is to set the wait count to $\frac{1}{4}$ of the Nodeslot count, this relies on the assumption workload is evenly distributed across functional units. In practice, overestimating the wait count leads to a high number of idle cycles, while underestimating it leads to a loss in throughput. It was found the optimal choice varies according to the input graph and node programming order due to the non-uniformity of node degrees. To overcome these challenges, the wait count can be dynamically optimized at run time by Host software.

Finally, another significant performance bottleneck was found to be the time taken to program node configuration into Nodeslots, with an average latency of 51 cycles. As shown in Algorithm 2, the testbench driver initially iterated through the list of pending nodes, waiting for resources to be freed in the event that no Nodeslots were available for the current node's precision. Nodeslots are chosen according to an arbitrary function, which can represent round-robin or find-first arbitration. Note that the nodeslots_free_mask is asynchronously asserted when a Nodeslot finishes computation.

Algorithm 2 Naive Nodeslot programming

Require: node_list $\mathcal{N}$, nodeslots_free_mask
  for node in $\mathcal{N}$ **do**
    while nodeslots_free_mask == '0 **do**
      pass
    end while
    chosen_nodeslot ← choose_nodeslot(nodeslots_free_mask)
    nodeslot_programming [chosen_nodeslot] ← node
    nodeslots_free_mask [chosen_nodeslot] ← 0
  end for

During simulation with the Pubmed graph, it was found that a high number of cycles were wasted waiting for **FIXED_8** Nodeslots to become free while **FLOAT_32** Nodeslots were already available. This latency was reduced by formulating an improved algorithm, as shown below. Here, nodes are separated into precision groups prior to the forward pass, and an element is taken from the appropriate queue every time a Nodeslot is free. Finally, latency was further reduced by omitting write transactions where the configuration matches the register's reset value, reducing the required number of AXI transactions.

Algorithm 3 Improved Nodeslot programming

Require: node_groups $\mathcal{N}_i$ $\forall i = 0...$PRECISION_COUNT, nodeslots_free_mask
  while True **do**
    if nodeslots_free_mask != '0 **then**
      chosen_nodeslot ← choose_nodeslot(nodeslots_free_mask)
      i ← chosen_nodeslot.precision
      nodeslot_programming [chosen_nodeslot] ← node_groups[i].head()
      nodeslots_free_mask [chosen_nodeslot] ← 0
    end if
  end while

At the time of writing, the Host software is made aware of finished Nodeslots via polling AXI read

transactions to the empty nodeslots mask in the NSB. As future work, Nodeslot programming latency can be reduced further by introducing an interrupt flow into the Node Scoreboard. Under this approach, an interrupt pulse is sent to the Host every cycle a Nodeslot transitions into EMPTY state, indicating the Nodeslot is ready for new work and bypassing the latency incurred by AXI transaction driving and arbitration.

8 Conclusion

This work presented the design, implementation, and performance evaluation of **AGILE**, a hardware accelerator tailored towards Graph Neural Networks. **AGILE** presents a novel architecture leveraging a dual-axis parallelism at the node and precision levels, providing a novel avenue for GNN acceleration. Its asynchronous programming model overcomes pipeline gaps in the typical double-buffering approach, owing to the non-uniform distribution of node degrees. Finally, an on-chip streaming Prefetcher unit was shown to lead to significant latency improvements.

A thorough evaluation of AGILE against conventional CPU and GPU counterparts demonstrates its superior performance across a diverse range of graph datasets. Inference latency on the Planetoid graphs exceeds GPU equivalents by up to $2.8\times$, significantly outperforming the CPU baseline and opening the door to future optimizations to increase the latency gap with GPUs on large graphs.

8.1 Future Work

At the time of writing, **AGILE** contains a robust set of features required for low-latency inference on Graph Convolutional Network (GCN) layers across a range of graph datasets. From October 2023, the author will undertake a PhD at Imperial College with a focus on GNN acceleration on multi-device FPGA clusters. The presented work will act as a starting point for this research direction, and the following extensions are planned to improve design capabilities.

8.1.1 Beyond Graph Convolutional Networks

Generalization to arbitrary Message-Passing Neural Networks (MPNNs) requires a small set of microarchitectural extensions. Equation 15 expresses the node-wise update law as a function of neighbour and edge embeddings (see Section 2.4.2 for further details). It can be seen that in the general case, concatenation of feature aggregations with the node embedding x_i^l may be required prior to transformation through the γ function. This can be achieved via a direct interface between the Prefetcher's Feature Bank and the Aggregation Buffer, with associated extensions to the Nodeslot state machine.

$$x_i^{l+1} = \gamma(x_i^l, \mathcal{A}_{j \in \mathcal{N}(i)}(\phi(x_i^l, x_j, e_{i,j}^l))) \tag{15}$$

Most importantly, outgoing messages are computed by an arbitrary function ϕ applied to the concatenation of each node's updated embedding with the updated neighbour and edge embeddings. Although on-chip pipelining of the transformation and Message Passing phases is desirable, this approach was deemed infeasible since temporal locality of neighbouring embedding updates is not guaranteed, meaning updated features must be written back to off-chip memory.

In the extended architecture, the computation of outgoing messages will be performed by a new Message Passing Engine (MPE), which is similar in structure to the FTE. Message computation requires the Host to program a dedicated Nodeslot with an asserted Message Passing flag. This indicates the Prefetcher to drive the MPE directly, bypassing the high-latency aggregation steps in the AGE.

Finally, computation of outgoing messages through the ϕ function requires concatenation with edge embeddings $e^l_{i,j}$ in the general case. This requires the implementation of an Edge Bank within the Prefetcher with a direct interface to the Message Passing Engine. The Edge Bank will contain a parametrizable number of Fetch Tags following the message fetching flow, i.e. streaming pointers into an Adjacency Queue with start addresses to be used for fetching the edge embeddings into a separate Embedding Queue.

8.1.2 Clock Gating

One of the primary benefits of inference on FPGA devices compared to GPU or CPU counterparts is their relatively low power consumption. However, due to the flexibility provided by **AGILE** in supporting a large number of numerical representations for weights and activations, significant portions of the circuit can be left idle depending on the application workload. Clock gating is a technique that has been widely deployed in ASIC and FPGA architectures to reduce dynamic power consumption by disabling the clock signal driving idle regions of a circuit. Within **AGILE**, clock gating can be used to dynamically power down unused arithmetic cores according to the layer configuration and Nodeslots programming.

The clock gating flow will be controlled by the Host application by writing to registers in each functional unit through the AXI-L protocol. Each unit contains a clock gating enable register, which, when asserted, instructs the control logic to gate the clocks if possible. Through additional fields, the Host specifies whether to gate the whole unit or apply more fine-grained gating. The CG_EN is a read-only register from the hardware perspective. The Host writes into an auto-clearing register to specify the desired gating strategy, which is unit dependent. Finally, the Host sees a read-only register with the real-time status of the clocks on the hardware. This is written by the accelerator in the case when gating has been requested by a section of the design, but the workload still requires that section to be active.

8.1.3 Node Halting

Inference computation on each node incurs a significant latency due to the off-chip memory access for incoming messages, distribution of messages across Aggregation Cores, matrix multiplication with learned parameters and the memory writeback mechanism for outgoing messages. In some cases, the Host may offload work onto **AGILE** before it is clear whether feature updates for a given node are required by the application. Once this decision is made, the Host would typically have to wait until the computation is finished for that node, i.e. all outgoing messages have been stored back into off-chip memory.

To overcome this latency problem, a Node Halting feature was proposed. This enables the Host to stop any pending work before reaching the Writeback stage. This is controlled by a set of auto-clearing **HALT_NODESLOT** registers in the NSB. When the NSB receives a halt pulse, the subsequent behaviour is dependent on the current state of the Nodeslot. When in AGGREGATION or TRANSFORMATION, the NSB will wait until the response is received from the AGE/FTE, signalling that the intermediate computations have been stored in the Aggregation/Transformation Buffers, respectively. At this point, the NSB sends a request to the relevant unit to dump any results pertaining to that Nodeslot, which clears space in the Buffers and increases the resource availability in subsequent stages of the pipeline. When in the FETCH_NB_LIST or FETCH_NEIGHBOURS states, the NSB requests the Prefetcher to drop any neighbour addresses or incoming messages associated with the Nodeslot after receiving the fetch done response. After some housekeeping, the Nodeslot state machine then transitions back to the EMPTY state.

Although node halting was found to be outside the scope of GNN acceleration, this feature presents the potential for significant latency improvements in real-time graph processing applications.

9 References

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 5 2017. ISSN 15577317. doi: 10.1145/3065386. URL https://dl.acm.org/doi/10.1145/3065386. pages 2

[2] Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 1:4171–4186, 10 2018. doi: 10.48550/arxiv.1810.04805. URL https://arxiv.org/abs/1810.04805v2. pages 2

[3] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran, and Zhenbin Wu. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 13(7), 4 2018. doi: 10.1088/1748-0221/13/07/P07027. URL http://arxiv.org/abs/1804.06913http://dx.doi.org/10.1088/1748-0221/13/07/P07027. pages 2

[4] Abdallah Moussawi, Kamal Haddad, and Anthony Chahine. An FPGA-Accelerated Design for Deep Learning Pedestrian Detection in Self-Driving Vehicles. 9 2018. doi: 10.48550/arxiv.1809.05879. URL https://arxiv.org/abs/1809.05879v1. pages 2

[5] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA*, pages 47–56, 2012. doi: 10.1145/2145694.2145704. URL https://dl.acm.org/doi/10.1145/2145694.2145704. pages 2

[6] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A White Paper on Neural Network Quantization. 6 2021. doi: 10.48550/arxiv.2106.08295. URL https://arxiv.org/abs/2106.08295v1. pages 2, 12

[7] Juanhui Li, Harry Shomer, Jiayuan Ding, Yiqi Wang, Yao Ma, Neil Shah, Jiliang Tang, and Dawei Yin. Are Graph Neural Networks Really Helpful for Knowledge Graph Completion? 5 2022. doi: 10.48550/arxiv.2205.10652. URL https://arxiv.org/abs/2205.10652v2. pages 2

[8] Marinka Zitnik and Jure Leskovec. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*, 33(14):i190–i198, 7 2017. doi: 10.1093/bioinformatics/btx252. URL http://arxiv.org/abs/1707.04638http://dx.doi.org/10.1093/bioinformatics/btx252. pages 2

[9] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. HyGCN: A GCN Accelerator with Hybrid Architecture. *Proceedings - 2020 IEEE International Symposium on High Performance Computer Architecture, HPCA 2020*, pages 15–29, 1 2020. doi: 10.48550/arxiv.2001.02514. URL https://arxiv.org/abs/2001.02514v1. pages 2, 1, 8

[10] Stefan Abi-Karam, Yuqi He, Rishov Sarkar, Lakshmi Sathidevi, Zihang Qiao, and Cong Hao. GenGNN: A Generic FPGA Framework for Graph Neural Network Acceleration. 1 2022. doi: 10.48550/arxiv.2201.08475. URL https://arxiv.org/abs/2201.08475v1. pages 2, 1, 9, 10, 16

[11] Shyam A. Tailor, Javier Fernandez-Marques, and Nicholas D. Lane. Degree-Quant: Quantization-Aware Training for Graph Neural Networks. 8 2020. doi: 10.48550/arxiv.2008.05000. URL https://arxiv.org/abs/2008.05000v3. pages 2, 6, 12

[12] Andry Alamsyah, Budi Rahardjo, and Kuspriyanto. Social Network Analysis Taxonomy Based on Graph Representation. 2 2021. URL https://arxiv.org/abs/2102.08888v1. pages 1

[13] Hsiang-Yun Wu, Martin Nöllenburg, and Ivan Viola. Graph Models for Biological Pathway Visualization: State of the Art and Future Challenges. 10 2021. URL https://arxiv.org/abs/2110.04808v1. pages 1

[14] Shoujin Wang, Liang Hu, Yan Wang, Xiangnan He, Quan Z. Sheng, Mehmet A. Orgun, Longbing Cao, Francesco Ricci, and Philip S. Yu. Graph Learning based Recommender Systems: A Review. *IJCAI International Joint Conference on Artificial Intelligence*, pages 4644–4652, 5 2021. ISSN 10450823. doi: 10.24963/ijcai.2021/630. URL https://arxiv.org/abs/2105.06339v1. pages 1

[15] Jonathan M. Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M. Donghia, Craig R. MacNair, Shawn French, Lindsey A. Carfrae, Zohar Bloom-Ackerman, Victoria M. Tran, Anush Chiappino-Pepe, Ahmed H. Badran, Ian W. Andrews, Emma J. Chory, George M. Church, Eric D. Brown, Tommi S. Jaakkola, Regina Barzilay, and James J. Collins. A Deep Learning Approach to Antibiotic Discovery. *Cell*, 180(4):688–702, 2 2020. ISSN 1097-4172. doi: 10.1016/J.CELL.2020.01.021. URL https://pubmed.ncbi.nlm.nih.gov/32084340/. pages 3

[16] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A Gentle Introduction to Graph Neural Networks. *Distill*, 6(9):e33, 9 2021. ISSN 2476-0757. doi: 10.23915/DISTILL.00033. URL https://distill.pub/2021/gnn-intro. pages 3

[17] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 9 2016. doi: 10.48550/arxiv.1609.02907. URL https://arxiv.org/abs/1609.02907v4. pages 4

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *Advances in Neural Information Processing Systems*, 2017-December:5999–6009, 6 2017. ISSN 10495258. doi: 10.48550/arxiv.1706.03762. URL https://arxiv.org/abs/1706.03762v5. pages 5

[19] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. URL http://graphchi.org. pages 8

[20] NJIT Advanced Computer System Design Laboratory. Systolic-Array Implementation of Matrix-By-Matrix Multiplication. URL http://ecelabs.njit.edu/ece459/lab3.php. pages 9

[21] Petar Veličković, Arantxa Casanova, Pietro Liò, Guillem Cucurull, Adriana Romero, and Yoshua Bengio. Graph Attention Networks. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 10 2017. doi: 10.48550/arxiv.1710.10903. URL https://arxiv.org/abs/1710.10903v3. pages 10

[22] Bingyi Zhang, Hanqing Zeng, and Viktor Prasanna. Hardware acceleration of large scale GCN inference. *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, 2020-July:61–68, 7 2020. ISSN 10636862. doi: 10.1109/ASAP49362.2020.00019. pages 17

[23] agalimberti/NoCRouter: RTL Network-on-Chip Router Design in SystemVerilog by Andrea Galimberti, Filippo Testa and Alberto Zeni. URL `https://github.com/agalimberti/NoCRouter`. pages 24

[24] AMBA AXI Protocol Specification. URL `https://developer.arm.com/documentation/ihi0022/latest/`. pages 32

[25] alexforencich/verilog-axi: Verilog AXI components for FPGA implementation. URL `https://github.com/alexforencich/verilog-axi`. pages 37

[26] Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Revisiting Semi-Supervised Learning with Graph Embeddings. *33rd International Conference on Machine Learning, ICML 2016*, 1:86–94, 3 2016. doi: 10.48550/arxiv.1603.08861. URL `https://arxiv.org/abs/1603.08861v2`. pages 40