2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-8711-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPS57955.2024.00084

Aurora: A Versatile and Flexible Accelerator for Graph Neural Networks

Jiaqi Yang
Electrical and Computer Engineering
George Washington University
Washington, DC, USA
Yang _ Jiaqi _ Cute@gwu.edu

Hao Zheng
Electrical and Computer Engineering
University of Central Florida
Orlando, Florida, USA
hao.zheng@ucf.edu

Ahmed Louri
Electrical and Computer Engineering
George Washington University
Washington, DC, USA
louri@gwu.edu

***Abstract*—Graph Neural Networks (GNNs) are pervasive across many application domains, driven by the growing demands to comprehend non-euclidean data. However, it remains a challenge to efficiently accelerate GNN applications due to varying graph data structures and diversified models. The non-euclidean data structure exhibits large variations in vertex and edge entities, and complex GNN models involve various computation characteristics and dataflows across different execution phases such as aggregation and vertex update. To address such diverse computation and communication demands, prior accelerators managed to tailor the compute engine for each GNN execution phase. Unfortunately, such limited adaptability could result in resource under-utilization and additional data movement.**

In this paper, we propose a versatile and flexible GNN accelerator that can efficiently facilitate the execution of various GNN models. The proposed architecture can be dynamically configured to multiple sub-accelerators, and each could be optimized for different GNN execution stages, including edge update, aggregation, and vertex update. To achieve this, the proposed architecture consists of four salient designs - a flexible processing element (PE) architecture, a unified Network-on-Chip (NoC), a degree-aware mapping, and a partitioning heuristic. The proposed unified processing element can be configured to support various fundamental GNN computations, such as vector-vector multiplication, vector-matrix multiplication, and scalar operations. The proposed NoC design can be dynamically configured to align with varying graph connectivity by dynamically bridging long-distance communications. The degree-aware mapping is proposed to avoid the unbalanced communication problem caused by high-degree vertices at the aggregation phase. Lastly, the proposed partitioning approach could efficiently allocate the computing resources to different GNN execution phases, improving the inter-phase parallelism. As such, the proposed design can increase hardware utilization with much-improved performance and energy efficiency. Simulation results show that our proposed Aurora accelerator achieves 85%, 66%, 47%, 28%, and 38% execution time reduction, and 89%, 77%, 42%, 69%, and 71% energy consumption reduction on average of multiple GNN dataset when compared to state-of-the-art accelerators HyGCN [1], AWB-GCN [2], GCNAX [3], ReGNN [4], and FlowGNN [5], respectively.

***Index Terms*—NoC, reconfigurable accelerator, Graph Neural Networks (GNNs)**

I. Introduction

Graph Neural Networks (GNNs) have been widely used to handle a variety of intricate scientific and engineering problems [6], [7], including but not limited to community detection [8], vertex classification [9], [10], point-cloud processing [11], social media recommendation systems [12], [13], and smart traffic [14], among others. Given its ability to comprehend graph data, a proliferation of GNN models has recently emerged, but it comes with a variety of computation requirements. For example, Graph Convolutional Networks (GCNs) rely on the feature vectors of vertices to capture the graph structure through aggregation operations, whereas the attention coefficients of Graph Attention Networks (GATs) require edge embeddings to record interactions between vertices. Mathematically, these vertex and edge activities are formulated in different forms - vector-vector, matrix-vector, and scalar operations.

TABLE I
GNN COVERAGE AND FEATURES OF **AURORA** IN COMPARISON WITH PRIOR WORKS

Accelerator	Models			Flexible operation in unified PE	Flexible dataflow in unified architecture	Flexible NoC	Message-passing
	C-GCN	A-GCN	MP-GCN				
AWB-GCN[2]	◑[1]	✗	✗	✗	✗	✗	✗
HyGCN[1]	◑[1]	✗	✗	✗	◑[2,4]	✗	✗
GCNAX[3]	◑[1]	✗	✗	✗	◑[3]	✗	✗
ReGNN[4]	◑[1]	✗	✓	✗	◑[4]	✗	✓
FlowGNN[5]	✓	✓	✓	✗	◑[2]	✗	✓
Aurora	✓	✓	✓	✓	✓	✓	✓

✓ Full support; ◑[1] Without edge embeddings; ◑[2] Without flexible execution phases workflow; ◑[3] Without pipeline flow; ◑[4] Without unified architecture; ✗ No support

Moreover, graphs do not adhere to a fixed size and pattern. Real-world graph data is typically comprised of millions or billions of vertices, each exhibiting a substantial variation in degree and connectivity. The irregular graph structure further adds complexity to efficient GNN acceleration. For example, each vertex needs to collect the feature vectors from its neighboring nodes. The high-degree vertices could cause one-to-many communication patterns, which inevitably need higher network bandwidth. On the other hand, irregular graph connectivity introduces long-distance communications. The combined problems could further lead to the adverse effects of an unbalanced communication workload, exacerbating the network performance.

While significant efforts [1]–[3], [5], [15]–[18] have been recently proposed to facilitate GNN execution, very few of them are efficient enough to address the mentioned challenges,

1530-2075/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPS57955.2024.00084

as shown in Table I. For example, HyGNC [1], GCNAX [3], and AWB-GCN [2] are designed to facilitate the computations of graph convolutional neural networks (GCNs). However, they abstract GCN computations as matrix multiplications and, as a result, do not support complex graph operations like edge updates. To support complex graph operations, FlowGNN [5] introduces a generic architecture to execute both graph and neural operations with message-passing. However, FlowGNN adopted heterogeneous edge and vertex compute engines to manage different GNN computation characteristics, leading to resource under-utilization and extra data movement between different compute engines. Similarly, ReGNN [4] presents a dynamic redundancy-eliminated neighborhood message passing to improve graph data locality. However, its performance is also restricted by the separate executions of graph and neural operations.

The crux of the mentioned problems is the current GNN accelerator architecture lacks enough flexibility to withstand varying computing and communication requirements. The computing units should handle various computation operations such as vector-vector, matrix-vector, and scalar operations. In addition, the communication fabrics should adapt to irregular graph connectivity and conventional dataflows (e.g., weight-stationary) [19]–[21]. To this end, we propose Aurora, a versatile and flexible GNN accelerator that can adapt to various GNN models with improved performance and energy efficiency. Specifically, the proposed Aurora can be dynamically partitioned into multiple sub-accelerators, adapting to a variety of communication and computation characteristics across different GNN execution phases. The main contributions of this paper are:

- We propose a versatile and flexible architecture that can support various GNN models with message-passing operations. Specifically, the proposed processing element (PE) architecture can be configured to support vector-vector, matrix-vector, scalar operations, etc. Furthermore, the proposed interconnect can bridge long-distance communication and support weight-stationary dataflow.
- We propose a degree-aware mapping policy, along with the flexible interconnect, that can alleviate the unbalanced communication load caused by high-degree vertices.
- We propose a partitioning algorithm that can efficiently allocate computing resources among different GNN execution phases. This can effectively improve the pipeline efficiency of GNN executions.

We implemented the proposed Aurora accelerator in RTL with the TSMC 40nm library. We also build a cycle-accurate simulator that models the microarchitectural behavior of each module with the degree-aware mapping strategy and partition algorithm for each GNN application. Evaluated on five real-world graph datasets, our simulation results show that Aurora achieves 85%, 66%, 47%, 28%, and 38% execution time reduction and 89%, 77%, 42%, 69%, and 71% energy consumption reduction on average when compared to HyGCN [1], AWB-GCN [2], GCNAX [3], ReGNN [4], and FlowGNN [5], respectively.

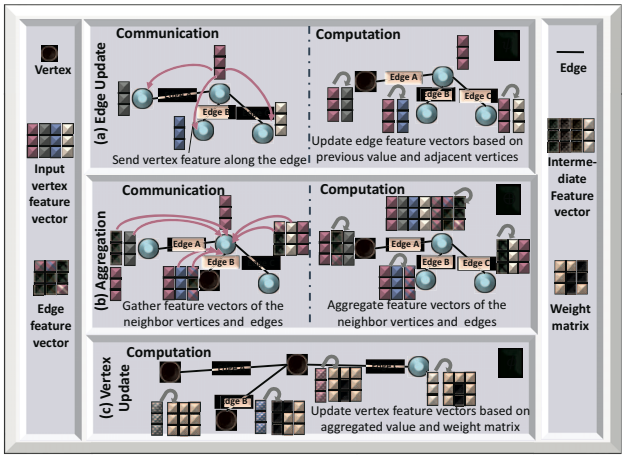

Fig. 1. Communication and computing abstraction of GNN processing phases: (a) Edge Update, (b) Aggregation, (c) Vertex Update

II. Background

Graph Neural Networks (GNNs) are a class of neural networks that are specifically designed to operate on graph-structured data. The basic idea is to use a message-passing scheme, where each vertex sends messages to its neighbors, and the messages are aggregated to compute an updated representation of the vertex [22]. Based on the most widespread pattern observed in GNN models [23], [24], abstract the execution model in a message-passing layer into three important stages: Edge Update, Aggregation, and Vertex Update. The execution stages are listed as follows and shown in Fig. 1:

- Edge Update: It updates the features for each edge by gathering the vertex features along the edges and computing the edge feature vectors based on their previous value and vertex features of adjacent vertices by an Edge Update function ψ, shown in Fig. 1 (a).
- Aggregation: This operation aggregates the neighbor information for each vertex by gathering the feature vectors of the neighbor vertices and edges and aggregating them to a single feature vector by an Aggregation function $\oplus$, shown in Fig. 1 (b).
- Vertex Update: It updates the feature vector for each vertex based on the aggregated feature vector (intermediate vector) and the weight matrix by a Vertex Update function ϕ, shown in Fig. 1 (c).

After iterations of Edge Update, Aggregation, and Vertex Update, a vertex is represented by its final feature vector, which captures the graph structure and feature information within the vertex's neighborhood. There are many different kinds of GNNs, their specific communication (message-passing) scheme and specific computation functions for Edge Update, Aggregation, and Vertex Update also vary.

GNNs generally fall into three categories based on the details of the Vertex Update function (ψ): Convolutional GNNs (C-GNNs), Attentional GNNs (A-GNNs), and Message-Passing GNNs (MP-GNNs). In short, in these three categories

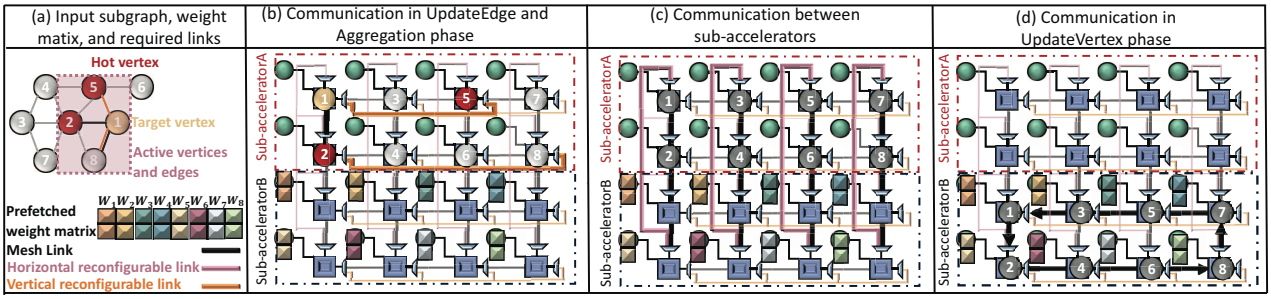

Fig. 2. An illustrative example of different communication requirements and NoC configuration for pipeline execution of Edge Update, Aggregation, and Vertex Update phases. (a) An input subgraph, prefetched weight matrix, and required links; (b) Communication pattern and NoC configuration of Edge Update and Aggregation phases in sub-acceleratorA; (c) Communication pattern between the sub-accelerators and its relative NoC configuration; (d) Communication pattern and NoC configuration of Vertex Update phase in sub-acceleratorB.

of GNN models, the Vertex Update function ψ is a fixed scalar coefficient (C-GNNs), a learnable function that returns a scalar coefficient (A-GNNs), or a learnable function that returns a vector coefficient (MP-GNNs).

A. C-GNN

1) Graph Convolutional Network (GCN): GCNs [25] operate by iteratively propagating information from neighboring vertices to update the vertex representations shown in Equation 1.

$$m_v^k = \sum_{u \in N(v) \cup v} \frac{x_u^{k-1}}{\sqrt{D_u \cdot D_v}}, \qquad x_v^k = ReLU(W^k m_v^k + b^k) \tag{1}$$

where m_v^k represents the aggregation results for vertex v at the k-th layer, x_u^{k-1} denotes the feature vector of vertex u (neighbor of vertex v) at the $k-1$-th layer, and D_v and D_u are the degree values of vertex v and vertex u, respectively. $ReLU$ denotes the activation function. W^k is the model weights, and b^k is the bias for the k-th layer.

2) Graph Isomorphism Network (GIN): GIN [26] was introduced as a framework for learning deep graph representations that are invariant to the graph's vertex ordering or isomorphisms. Equation 2 shows the overall computation for k-th layer of a multi-layer GIN model.

$$m_v^k = (1+\epsilon^k)x_u^{k-1} + \sum_{u \in N(v)} x_v^{k-1}, \qquad x_v^k = MLP^k(m_v^k) \tag{2}$$

where ϵ^k denotes a learnable parameter, and MLP^k represents the multilayer perceptron for the k-th layer.

B. A-GNN

Attention-based GNN models [27], [28] implies the attention mechanism to achieve more accurate predictions by learning a dynamic and adaptive local summary of the neighborhood. Equation 3 shows the overall computation for k-th layer of a multi-layer Attention-based GNN model.

$$m_v^k = \sum_{u \in N(v)} ((x_v^{k-1})^T \cdot x_u^{k-1}) x_u^{k-1}, \qquad x_v^k = SoftMax(W^k m_v^k) \tag{3}$$

where $SoftMax$ function used as the last activation function to normalize the output of a network to a probability distribution over predicted output classes.

C. MP-GNN

1) Gated Graph Convolutional Networks (G-GCN): G-GCN [29] incorporates gating mechanisms to regulate the information flow during message passing among nodes in a graph as shown in Equation 4. By focusing on important neighbors and discarding irrelevant information selectively, G-GCN significantly enhances the expressive capacity of GCN [30].

$$m_v^k = \sum_{u \in N(v)} \sigma(W_u^k x_u^{k-1} + W_v^k x_v^{k-1}) \odot x_u^{k-1}, \qquad x_v^k = ReLU(W^k m_v^k) \tag{4}$$

where W_u^k and W_v^k are weights for vertex u and vertex v, respectively. $\odot$ denotes the element-wise multiplication.

2) GraphSage - Pool: GraphSage-Pool [10], [31], shown in Equation 5, selects representative vertices from the graph based on certain criteria. The selected nodes are then used to construct a pooled graph, which serves as a condensed version of the original graph. This pooling process helps to reduce the computational complexity of subsequent operations while still preserving the essential information of the graph.

$$m_v^k = Concat(max(\{\sigma(W_{pl} x_u^{k-1} + b))r\}, x_v), \qquad x_v^k = ReLU(W^k m_v^k + b^k) \tag{5}$$

where $Concat$ denotes the concatenation operation, max represents the element-wise max operator, σ is the non-linear activation function and W_{pl} is the pooling weights.

III. Proposed Accelerator Architecture

In this paper, the goal of our proposed Aurora accelerator is to support a wide range of GNN models, including GAN, GCN, and GIN, among others. The proposed design can adapt to distinct computation and communication characteristics. As shown in Fig. 2, Aurora can be dynamically partitioned into multiple sub-accelerators, facilitating multiple GNN execution phases in one unified architecture. Specifically, the proposed PE architecture can be adaptive to distinct computations such as matrix-vector, vector-vector, and scalar operations, and the communication fabrics can be dynamically configured

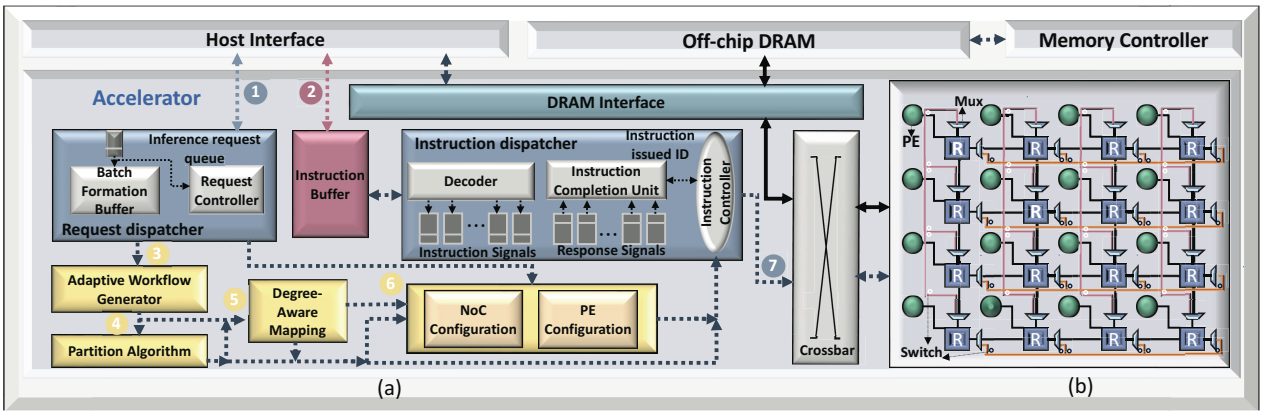

Fig. 3. Overall architecture of **Aurora**: (a) proposed controllers including request dispatcher, instruction buffer, instruction dispatcher, adaptive workflow generator, degree-aware mapping algorithm, partition algorithm, NoC and PE configuration unit; (b) a 4 × 4 reconfigurable NoC design.

to reduce long-distance communication and support various dataflows. For example, bypassing links could be configured to facilitate the injection of high-degree vertices at the aggregation phase. Multiple rings could be configured to support weight-stationary dataflow for vertex update.

A. Proposed Aurora Accelerator Design

The proposed Aurora design features two unique designs - unified PE architecture and adaptive interconnect. The proposed PE design can support a variety of computation characteristics such as matrix-vector, vector-vector, and scalar operations. Moreover, Aurora's flexible interconnect can handle irregular graph connectivity (e.g., aggregation) and weight-stationary dataflow (e.g., vertex update). Overall, as shown in Fig. 3, the proposed Aurora accelerator consists of a request dispatcher, an instruction buffer, an instruction dispatcher, an adaptive workflow generator, a partitioning algorithm, a degree-aware mapping algorithm, PE and NoC configuration units, a versatile PE array interconnected by a flexible interconnect.

The host (e.g., CPU) sends requests to the request dispatcher and stores instructions in the instruction buffer. The request dispatcher then sends the running model and graph data. The graph data is stored using compressed sparse row (CSR) format, and its metadata will be used as auxiliary information to determine the workload mapping and to decide the resource partitioning for different GNN phases. The mapping and partitioning decisions will ultimately be sent for NoC and PE configurations.

To increase memory bandwidth, we implement a crossbar between the DRAM interface and processing elements (PEs). As such, the data can be distributed to different rows of PEs. The PE array adopts a distributed buffer design, in which an array of PEs are interconnected by a flexible NoC. Each PE consists of an array of flexible multiply–accumulate (MAC) units. The detailed architecture of PE and interconnect will be described in what follows.

B. Flexible NoC Design

Unlike communication patterns in conventional DNN acceleration, complex GNN models rely on message-passing

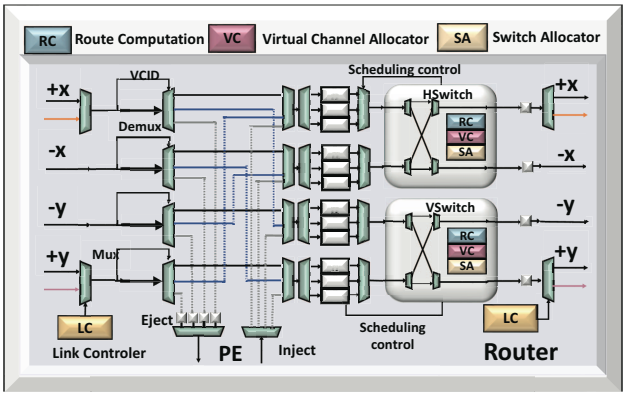

Fig. 4. Microarchitecture of reconfigurable router

to perform edge and vertex operations. Even though the message-passing has a wide applicability to generic GNN applications, it complicates the on-chip communication patterns. The communication pattern is subject to irregular graph connectivity and large variations in vertex degree. In particular, two problems arise - long-distance communication and an unbalanced communication workload. Consequently, certain NoC regions are flooded with a higher traffic load due to high-degree vertices, as shown in Fig. 2 (b). Moreover, the vertex update still adopts conventional dataflows, in which the weight matrix is distributed to multiple PEs, as shown in Fig. 2 (d). The feature vectors need to be accumulated across multiple PEs.

To address the mentioned challenges, the network topology needs to align with distinct graph connectivity and dataflows. In other words, this requires NoC to increase network bandwidth for high-degree vertices and provide dynamic link connectivity. As such, we propose a novel flexible NoC which is built upon conventional mesh topology. One bi-directional bypassing link is placed at each row and column of the network, and each bypassing link consists of multiple link switches (e.g., transistor). Consequently, the bypassing link can be segmented into multiple short links with varying lengths, as shown in Fig. 2. Those links will be used to bridge long-distance communications or be used as wrap-up links for ring topology.

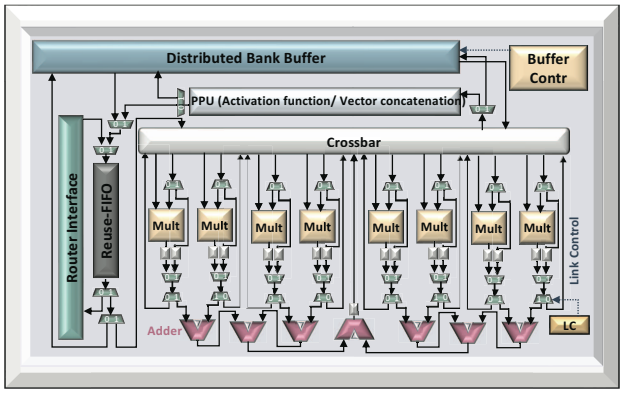

Fig. 5. Microarchitecture of reconfigurable PE

C. Proposed Flexible Router

The proposed router aims to handle the high bandwidth demanded by high-degree vertices at the aggregation phase, and to support bypassing links to reduce communication distances.

The proposed router is built upon conventional router architecture, including five key components: the Route Computation unit (RC), the Virtual Channel Allocator (VA), the Switch Allocator (SA), the virtual channel (VC) buffers, and the crossbar. To reduce the cost of the crossbar, the proposed router uses a two-stage design with horizontal and vertical switches. The horizontal and vertical switches can be decomposed to support ring topology, which is commonly used for DNN dataflows. Both switches can be combined as an all-to-all crossbar.

Furthermore, we add a few muxes at +x and +y, connecting to both mesh links and bypassing links. The link controller is used to determine if the bypassing link should be segmented and connected to the router. Given the limited wire budget, it is impractical to implement a bundle of bypassing links at each row or column. Considering the power-law distribution of real-world graphs, each graph partition could only have a few high-degree vertices. Consequently, only a few bypassing links are needed, but it requires a strategy to avoid the mapping of multiple high-degree vertices on the same row or column. This mapping strategy will be discussed in Section-IV.

D. Proposed Unified PE Architecture

The proposed PE architecture is to support the distinct computations at various GNN phases - edge update, aggregation, and vertex update. The key computation patterns of these stages further depend on the selected GNN model, which is summarized in Table 2. To increase resource utilization, it is beneficial to design a unified PE architecture capable of handling such diverse computation needs. This will eliminate unnecessary data movements between CPU, GPUs, and different dedicated compute engines. The proposed flexible PE architecture is shown in Fig. 5, which consists of a distributed bank buffer, a router interface, a reuse First-in-First-Out (FIFO) buffer, a Post Processing Unit (PPU), a buffer controller, multipliers, adders, and a reconfigurable interconnect between multipliers and adders.

TABLE II
REQUIRED OPERATIONS EXISTING IN DIFFERENT EXECUTION PHASES OF VARIOUS GNN MODELS. $Scalar$ DENOTES SCALAR COEFFICIENT, V MEANS VECTOR, M DENOTES MATRIX, $\times$ MEANS MULTIPLICATION, $\cdot$ DENOTES DOT PRODUCTION, $\odot$ MEANS ELEMENT-WISE PRODUCTION, $\sum$ DENOTES ACCUMULATION, α MEANS ACTIVATION FUNCTION, AND $||$ DENOTES CONCATENATION.

Execution Phase		Edge Update	Aggregation	Vertex Update
C-GNN	GCN [25]	$Scalar \times V$	$\sum V$	$M \times V$, α
	GraphSAGE -Mean [10]	Null		$M \times V$
	GIN [26]	Null		$M \times V$
	CommNet [32]	Null		$M \times V$
A-GNN	Vanilla-attention [27]	$Scalar \times V$, $V \cdot V$		$M \times V$, α
	Attention-based GNNS [28]	$Scalar \times V$, $V \cdot V$		$M \times V$, α
MP-GNN	G-GCN [29]	$M \times V$, $V \odot V$, α		$M \times V$,α
	GraphSAGE -Pooling [10]	$M \times V$,α		$M \times V$, $V\|\|V$, α
	EdgeConv-1 [33]	$M \times V$		Null
	EdgeConv-5 [33]	$M \times V$,α		Null

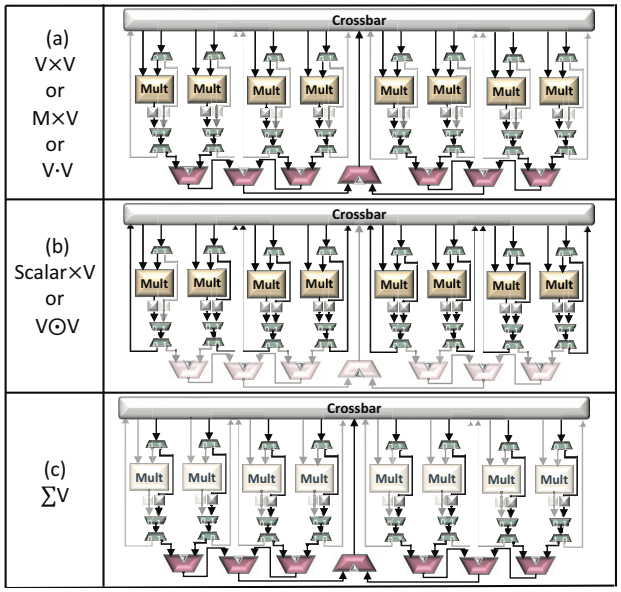

Fig. 6. Typical configuration examples of reconfigurable PE architecture to support various GNN operations. (a) Example of PE architecture configuration that can support $V \times V$, $M \times V$, $V \cdot V$ operations; (b) Example of PE architecture configuration that can support $Scalar \times V$ and $V \odot V$ operations; (c) Example of PE architecture configuration that can support $\sum V$ operation. ($Scalar$ denotes scalar coefficient, V means vector, M denotes matrix, $\times$ means multiplication, $\cdot$ denotes dot production, $\odot$ means element-wise production, $\sum$ denotes accumulation.)

The distributed bank buffer is used to increase the memory bandwidth to accommodate the random memory access caused by graph irregularity. The buffer controller sends the data to the MAC array via a crossbar. The MAC array can be reconfigured to process the vector-vector multiplication ($V \times V$), matrix-vector multiplication ($M \times V$), and vector-vector dot production ($V \cdot V$), as shown in Fig. 6. Such flexibility is enabled by the reconfigurability of the datapath. For example, multipliers

are paired and then connected to one adder, and adders are further connected sequentially for accumulation, as shown in Fig. 6 (a). On the other hand, for the scalar operations, a constant will be loaded to the multiplier, and the data will be written back to buffers without any accumulation operations, as shown in Fig. 6 (b). Lastly, for the accumulation operations, all the multipliers and adders will be bypassed, as shown in Fig. 6 (c).

To increase data reuse further, we need to enable inter-PE data exchange. To support this, we implement a reuse FIFO. The reuse FIFO acts as a double buffer [34] to store intermediate feature vectors received from neighboring PEs at the vertex update phase and updated edge feature vector received at the aggregation phase. The non-linear activation function and/or vectors concatenation are performed in the PPU, if necessary, before writing the output feature to the distributed bank buffer. The multipliers and adders are connected by reconfigurable interconnect in order to support multiple operations.

E. Walk-through Example

As illustrated in Fig.3 (a), the host (e.g., CPU) sends requests to the request dispatcher (❶) and loads instructions to the instruction buffer (❷). The control unit then dispatches GNN model information, along with graph metadata (e.g., row and edge indices in CSR format) to the preprocessing unit. To decide on mapping and partitioning, the auxiliary information will be processed through multiple units, including the adaptive workflow generator (❸), partition algorithm (❹), degree-aware mapping algorithm (❺), NoC, and PE configuration unit (❻).

Specifically, the Adaptive Workflow Generator (❸) decides the workflow of the running GNN model, such as execution phases and operation types. The resultant workflow information, coupled with graph metadata, is transmitted to the Partition Algorithm (❹) for determining resource partitioning across different GNN execution phases. The generated resource partition strategy, workflow information, and graph metadata are then forwarded to the Degree-Aware Mapping Algorithm (❺).

Furthermore, the generated workflow, resource partition strategy, workload mapping scheme, and graph metadata are dispatched to the NoC and PE Configuration Unit (❻). This unit determines the NoC configuration for different execution phases based on the communication requirements within and between sub-accelerators, as illustrated in Fig.2 (b-d). Afterward, PEs are configured depending on the specific computation requirements of sub-accelerators. Once the configuration is completed, the instruction dispatcher start issuing instructions as conventional accelerators (❼).

IV. Proposed Degree-aware Mapping

Even though the proposed NoC architecture can increase the network bandwidth for high-degree vertices, the mapping of high-degree vertices is also crucial. This is because only one bypassing link is available at each row or column given the cost concern. However, the communication patterns are

Algorithm 1: Degree-aware Mapping Algorithm

Input : Structure of graph inputs:
$G = \{V, E\}$, A input graph;
V and E are sets of vertices and edges.
n, Numbers of vertices in G; $|V| = n$.
$N(i)$, Neighbors of vertex i.
$D(i)$, The degree of a vertex i.
Input : Features of spatial accelerator:
$K \times K$, Layout of the PE array;
K is number of row and column of PE array.
C_{PE}, Buffer capacity of PE
Output: High-degree Vertex{}, S_PE{}

1 **Procedure** *S_PE Identification*
2 **Function** *Queen(int $k = 1$)*
3 **if** $k <= K$ **then**
4 **for** $a \in [1, K]$ **do**
5 /* Check N-Queen law */
6 **if** *canPlace(k,a)* **then**
7 $S\_PE\{\}$ ←PE[k][a];
8 $Queen(k + 1)$;
9 **end**
10 **end**
11 **end**
12 **return** $S\_PE\{\}$;
13 **Procedure** *High-Degree Vertex Identification*
14 $N_{S\_PE} \leftarrow \sum S\_PE\{\}$;
15 N_{HN} ←N$_{S_PE} \times C_{PE}$;
16 **Function** *Sort(int $D(i)$, N_{HN}, $nhn = 0$)*
17 Sort the N_{HN} vertices with the Maximum degree;
18 d ←Min(Degree of the sorted vertices);
19 **for** $i \in [1, n]$ **do**
20 **if** $D(i) \geq d$, $nhn \leq N_{HN}$ **then**
21 $HighdegreeVertex\{\}$←i;
22 $nhn++$;
23 **end**
24 **end**
25 **return** $HighdegreeVertex\{\}$;

subject to the graph connectivity - vertex connectivity and degree variation. Based on hashing-based mapping, high-degree vertices are very likely to be mapped to the same row or column, leading to resource contention. As such, we need to map high-degree vertices to different rows or columns.

Given this, we propose a mapping algorithm leveraging the N-Queen algorithm [35], in which high-degree vertices are mapped to disjoint rows or columns. Typically, real-world graphs are large, exceeding the on-chip memory capacity. We tile the large graph into several subgraphs based on on-chip memory size. We preprocess the graph metadata (e.g., row and edge indices in CSR format) to understand the edge and degree quantities, and the processed metadata will be used for deciding mapping. In other words, the mapping algorithm will be performed before the execution of each subgraph. After mapping a subgraph to the PE array, the next subgraph starts being loaded from DRAM to overlap the latency. The frequency of executing the mapping algorithm depends on the number of subgraphs. To be specific, We use Algorithm 1 to illustrate the details of the degree-aware mapping strategy. As shown in Algorithm 1 (lines 1-12), we need to identify the suitable PE locations ($S\_PE\{\}$) in order to map the high-

degree vertices. We use the N-Queen algorithm, in which no two PEs that process high-degree vertices (S_PE) can be placed in the same row, column, or diagonal. To reduce the algorithm complexity, we only select a fixed identification pattern for S_PE, in which each row only accommodates one S_PE. Afterward, we also need to determine the high-degree vertices, as shown in Algorithm 1 (lines 13-25). The vertices are first sorted based on the degree number, in which top N_{HN} vertices will be determined as high-degree vertices, N_{HN} is determined by the buffer capacity of PE (C_{PE}) and the number of S_PE (N_{S_PE}). For a $K \times K$ PE array, N_{HN} is $(K-1) \times C_{PE}$. Please note that the PE array for the aggregation phase varies according to the specific workload and model. Each row or column is only allowed to host one S_PE to process high-degree vertices. Furthermore, the bypassing links will be used to bridge the longest communications for each high-degree vertex. The sorted high-degree vertices will be mapped sequentially using a hashing-based method. The low-degree vertices will be mapped to PEs with available resources sequentially. As such, the overall algorithm complexity of the proposed method is $N \cdot log(N) + N$. Notably, the execution of the mapping algorithm can overlap with the GNN computations from the previous subgraph.

V. Proposed Partition Algorithm

As mentioned, GNN models are typically composed of multiple execution phases. Given distinct communication and computation characteristics, prior works mostly adopt a heterogeneous design to handle various GNN execution stages, in which multiple dedicated compute engines are designed. However, the computation load of various phases depends on the graph structure, feature size, and model. This could lead to resource under-utilization, given the fixed resource capacity at the design time. For example, for the EdgeConv model, the vertex update is not needed. Consequently, the vertex update engine could be underutilized in prior accelerators. On the other hand, for GIN models, edge update is not required, and so are the computing units. Furthermore, additional storage (e.g., global buffers) is needed to temporarily store the intermediate results between execution phases.

Given the proposed unified architecture, Aurora can improve resource utilization while avoiding additional storage between different execution phases. However, it requires a resource partitioning algorithm to allocate enough computing resources. The key idea is to balance the execution time between various GNN phases, improving the pipeline efficiency. To address this issue, we propose a partition algorithm 2 in support of various GNN models. The partitioned resources will be formed as two sub-accelerators, and each sub-accelerator will be configured for different computations and dataflows. The partitioned sub-accelerators are tailored to support diverse GNN computation and communication characteristics. In particular, the message aggregation stage involves irregular communication, while the neural network computation stage entails regular data movement. Consequently, two sub-accelerators are employed to address these distinct stages. The partitioning algorithms

Algorithm 2: Aurora Partition Algorithm

Input : Structure of graph inputs:
$G = \{V, E\}$, A input subgraph;
V and E are sets of vertices and edges.
m, Number of edges in $G; |E| = m$.
E_f, Number of edge features.
O_{ue}, Number of operations in Edge Update Phase.
O_a, Number of operations in Aggregation Phase.
O_{uv}, Number of operations in Vertex Update Phase.
Input : Features of spatial accelerator:
P, Number of PE.
$Flops$, Operations per second of PE.
Output: Partition Strategy (a,b)

1 // Calculating the computation time of sub-accelerator A
2 **Procedure** *Computation Time of the Sub-accelerator A*
3 $T_A = Max\{AComp_1, AComp_2\} + AComp_3$;
4 $AComp1 = \frac{O_{ue}}{a \times Flops}$;
5 $AComp2 = \frac{O_r - E_f \times m}{a \times Flops}$;
6 $AComp3 = \frac{E_f \times m}{a \times Flops}$;
7 **return** T_A;
8 // Calculating the Computation time of the sub-accelerator B
9 **Procedure** *Computation Time of Sub-accelerator B*
10 $T_B = \frac{O_{uv}}{(P-a) \times Flops}$;
11 **return** T_B;
12 // Generate the partition strategy with high hardware utilization
13 **Procedure** *Partition Strategy Generation*
14 **for** $a \in [0, P]$ **do**
15 $Diff_a = \|T_A - T_B\|$;
16 **end**
17 Find the minimal $Diff_a$;
18 **return** a
19 $b = P - a$;
20 **return** *Partition Strategy (a,b)*

are triggered by the arrival of a new sub-graph or a GNN layer. These two sub-accelerators are further connected to support the pipeline execution without the extra buffers to store intermediate results. Please note that only one accelerator will be formed if vertex updates are not required. If edge updates are not necessary, GNN execution can be initiated with aggregation, and set $AComp1$ to 0.

Specifically, the partitioning algorithm consists of the following three steps: (1) time estimation for Edge Update and Aggregation phases (Algorithm 2 Line 2-7), (2) time estimation for Vertex Update phase (Algorithm 2 Line 9-12), and (3) the partitioning strategy that can minimize the stall between execution phases and maximum the resource utilization (Algorithm 2 Line 13-20).

First, the Edge Update and Aggregation are performed independently and sequentially. Since both stages exhibit the same communication patterns, they are running on the same architecture. In this paper, we use sub-accelerator A to indicate the architecture configured for both edge update and aggregation. After that, the aggregated edge feature will be calculated and updated. To estimate the execution time for edge update and aggregation (T_A), we can use the edge information, such

as edge count and feature size. It includes the maximum of computation time for Edge Update ($Acomp_1$) and the computation time of the Aggregation phase ($Acomp_2$), and the computation time of edge update aggregation ($Acomp_3$). The computation time for Edge Update ($Acomp_1$) is estimated by using the number of operations in Edge Update (O_{ue}), the number of PEs assigned to sub-accelerator A (a), and operation per second of PE ($Flops$), shown in Algorithm 2 Line 4.

The computation time for the Aggregation phase ($Acomp_2$) is estimated by using the number of operations in the Aggregation phase (O_a), the number of edge features (E_f), the number of edges (m), the number of PEs assigned to sub-accelerator A (a), and operation per second of PE ($Flops$),shown in Algorithm 2 Line 5. The computation time for edge aggregation ($Acomp_3$) is estimated by using the number of edge features (E_f), number of edges (m), the number of PEs assigned to sub-accelerator A (a), and operation per second of PE ($Flops$), as shown in Algorithm 2 Line 6.

Second, The partition algorithm estimates the computation time for sub-accelerator B (T_B), which is used for edge update only. The computation time for Vertex Update is related to the number of operations in the Vertex Update phase (O_{uv}), the number of PEs assigned to sub-accelerator B ($P - a$), and operation per second of PE ($Flops$), as shown in Algorithm 2 Line 10. Finally, by finding the optimal a that minimizes the difference ($Diff_a$), a partitioning strategy is generated that can minimize stalls between execution phases and maximize resource utilization. The execution of the partition algorithm can also overlap with the GNN computations from the previous subgraph.

VI. Evaluation

A. Evaluation Setup

Accelerator Simulator : We built a cycle-accurate simulator to measure the performance of the proposed Aurora accelerator. In order to obtain execution time results, the simulator monitors the number of arithmetic operations and the number of accesses to each memory hierarchy, taking the degree-aware mapping algorithm, partition algorithm, and system configuration parameters into account. The number of arithmetic operations is used to calculate the computation time, while the number of accesses to each memory hierarchy is used to calculate the communication time.

The off-package communication time is obtained from the DRAMSim2 simulator [36]. The overall execution time is derived by adding up the computation time, the on-package communication time, and the off-package communication time, considering the overlap caused by the buffering of the distributed buffer and other memory hierarchies.

The simulator counts the required amount of on/off-chip communications and computations, which is used to estimate the related energy consumption according to [37]. Additionally, to accurately estimate the power and area consumption, we used the Synopsys Design Compiler with the TSMC 40 nm standard library to synthesize and generate the waveform activity file to capture the dynamic switching activity of the logic gates. Afterward, we used Synopsys PrimeTime PX with the waveform activity file to measure the dynamic and static power consumption. Specifically, we analyzed most of the accelerator components, including instruction Dispatcher, PEs, Task Control, partitioning and mapping algorithms, distributed buffers, and NoCs.

Accelerator Modeling : We implemented the proposed design including 32 $\times$ 32 PEs interconnected by a flexible NoC. Each PE consists of a distributed bank buffer, router interface, FIFO buffer, post-processing unit(PPU), buffer controller, multiplier, adder, and required logic. The on-chip frequency of the proposed accelerator is 700MHz. The distributed bank buffer capacity of each PE is 100KB. The proposed partition and mapping algorithms are performed before each subgraph undergoes execution. The frequency of applying these algorithms depends on the number of subgraphs, and the entire process can seamlessly overlap with the concurrent execution of GNN computations.

Baselines : We compare Aurora with five GNN accelerators (HyGCN [1], AWB-GCN [2], GCNAX [3], ReGNN [4], and FlowGNN [5]). To evaluate the efficiency of the degree-aware mapping, we use CGRA-ME [38], a popular open-source Coarse-Grained Re-configurable Arrays (CGRA) compilation framework. We select CGRA-ME because it allows flexible modeling of different spatial accelerators. Aurora is compared with the mapping approach from CGRA-ME: hashing-based mapping strategy. The baseline accelerators are scaled to be equipped with the same number of multipliers and DRAM bandwidth as Aurora. We uniformly use double precision for all accelerators to provide a fair comparison.

As HyGCN uses a tandem-engine architecture consisting of SIMD cores for the aggregation phase and systolic modules for the combination phase, the multipliers are divided into two groups in a ratio of 1:7 for the two engines according to its original configuration. We also resized the baseline accelerators to be equipped with the same on-chip storage capacity. For example, we simulated the baseline accelerators with 100 MB on-chip storage.

Benchmark: Regarding computation variations in GNN models, we use the following PolyBench benchmark operators [39] and nonlinear activation functions for the execution phases of generic GNN models:(a) Edge update phase: gramschmidt kernel (Gram-Schmidt Decomposition), mvt kernel (Matrix Vector Production), gemver kernel (Vector Addition), gesummv kernel (Vector Vector Multiplication), and nonlinear activation function (ReLu); (b) Aggregation phase: gemver kernel (Vector Addition); (c) Vertex update phase: mvt kernel (Matrix Vector Production) and nonlinear activation function (ReLu).

In the Edge Update phase, the gramschmidt kernel (Gram-Schmidt Decomposition) is employed to ensure that the resulting features are orthogonal, minimizing redundancy and capturing distinct information. The mvt kernel (Matrix Vector Production) is employed for the multiplication of the weight matrix associated with neighboring vertex feature vectors.

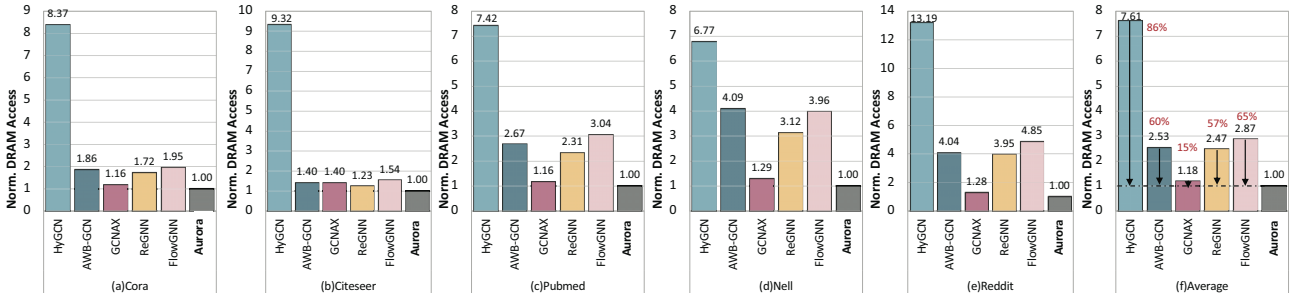

Fig. 7. Normalized DRAM accesses for baseline architectures and our proposed accelerator, normalized to the DRAM accesses of the proposed accelerator.

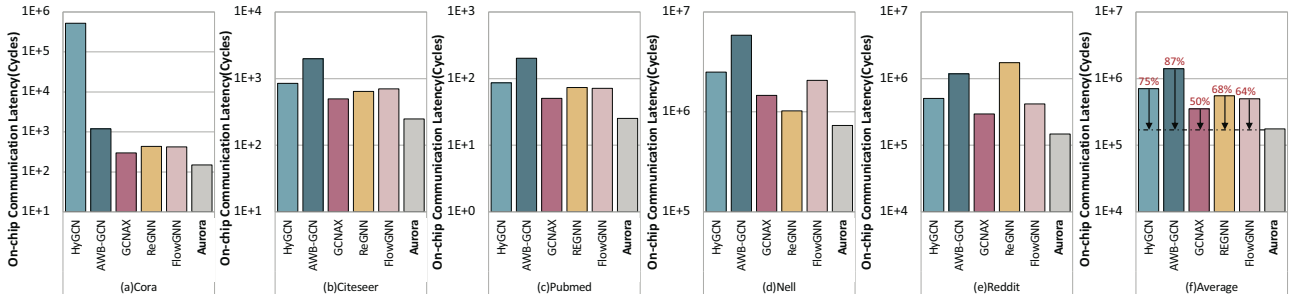

Fig. 8. On-chip communication latency for baseline architectures and our proposed accelerator.

Additionally, the gemver kernel (Vector Addition) is applied to aggregate the weighted vertex feature vectors calculated in the mvt kernel (Matrix Vector Production). The gesummv kernel (Vector Vector Multiplication) is employed to update edge features in which vectors are obtained from the gemver kernel and the feature vectors from neighboring vertices. Finally, a ReLU function is used as the activation function.

In the Aggregation phase, the gemver kernel (Vector Addition) is used to accumulate all neighboring feature vectors. In the Vertex Update phase, the mvt kernel (Matrix Vector Production) is utilized for the multiplication of the weight matrix associated with the accumulated feature vector obtained by the gemver kernel (Vector Addition) in the Aggregation phase. The Vertex Update phase is also finalized by a ReLU function, which is used as the last activation function.

Datasets: We use Cora, Citeseer, Pubmed, Nell, and Reddit as the datasets [10], [40], [41]

B. Off-chip DRAM Access Analysis

Fig. 7 illustrates the normalized DRAM access volume of both the baseline and the proposed accelerator for each dataset. It is evident that the proposed accelerator consistently outperforms the baselines. The baselines, unfortunately, fail to make full use of the on-chip buffer capacity. This limitation hampers their ability to minimize DRAM accesses.

The DRAM access reduction varies across the datasets. The proposed design achieves 86%, 60%, 15%, 57%, and 65% DRAM access reduction on average for each dataset when compared to baselines, respectively. To fully utilize the on-chip buffer capacity, the proposed accelerator selectively preloads the weight matrix and edge embedding data (if needed) into specific parts of the accelerator (referred to as related sub-accelerators). This approach differs from AWB-GCN, GCNAX, and FlowGNN, in which the weight matrix needs to be duplicated in all processing elements. Moreover, the proposed design can directly transfer the output feature vectors from sub-accelerator A to sub-accelerator B without the need for any storage. In contrast, AWB-GCN, GCNAX, and FlowGNN do not take advantage of this opportunity. HyGCN and ReGNN can also store weight matrices in specific heterogeneous sub-accelerators. However, their fixed heterogeneous sub-accelerators (or engines) often suffer from under-utilized memory and computing resources when dealing with various GNN models. HyGCN and ReGNN fail to fully harness on-chip data reuse opportunities due to the underutilization of memory resources, leading to increased DRAM accesses.

C. On-chip Communication Analysis

Due to the inherent complexity of graph structures, GNN models often encounter irregular and intricate communication patterns. Various GNN models demand the integration of degree-aware mapping algorithms, dynamic partition algorithms, and efficient hardware support to address these issues effectively.

Fig. 8 presents the on-chip communication latency for both the baseline and the proposed accelerator across different datasets. The measurement is based on the total number of on-chip communication cycles, with the proposed accelerator implementing degree-aware mapping algorithms, dynamic partition algorithms, and efficient hardware support.

Overall, the proposed design achieves 75%, 87%, 50%, 68%, and 64% on-chip communication latency reduction on average for each dataset when compared to baselines respectively. The on-chip communication latency is affected

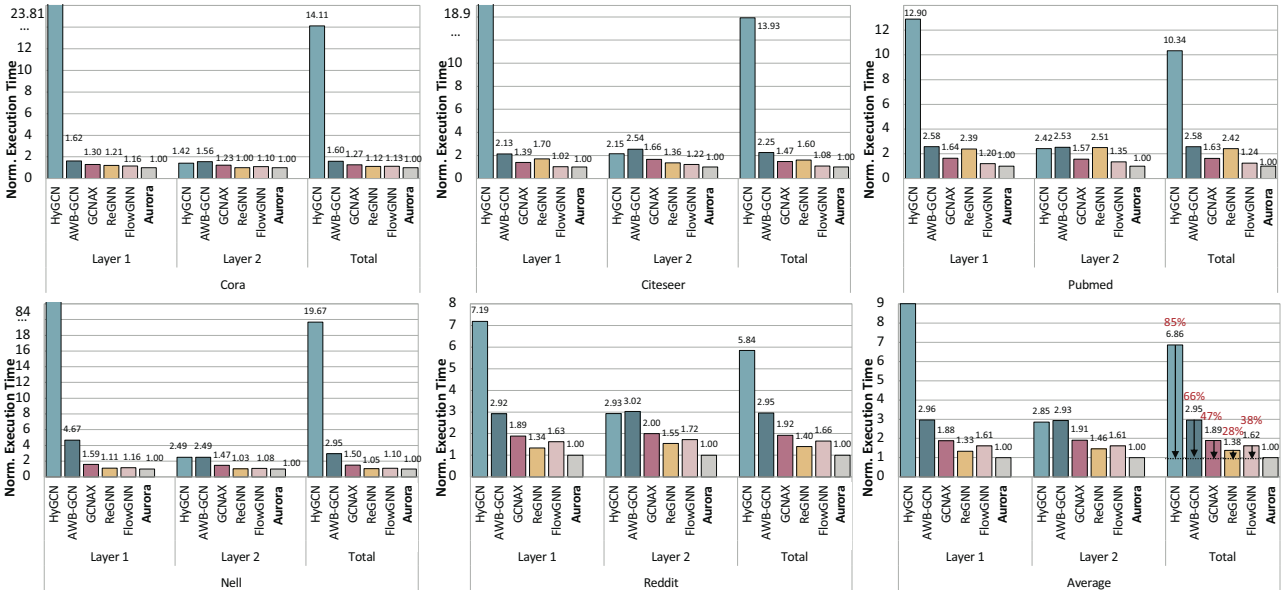

Fig. 9. Normalized execution time for each layer in multiple GNN datasets for baseline architectures and our proposed accelerator, normalized to the execution time of the proposed accelerator.

by communication amount, hop count, and efficient on-chip bandwidth.

The reasons behind the reduction in on-chip communication latency in the proposed design can be categorized into mutltiple key aspects. First, the inclusion of the degree-aware mapping algorithm effectively mitigates communication contention, leading to a reduction in on-chip communication latency. Second, the use of a reconfigurable architecture in the proposed accelerator provides robust support for diverse communication requirements while simultaneously decreasing the average hop count.

D. Performance Analysis

Fig. 9 shows the performance of the proposed design and the baselines (Baselines use hashing-based mapping strategies while the proposed accelerator provides degree-aware mapping algorithm, dynamic partition algorithm, and efficient hardware support).

The proposed accelerator achieves 85%, 66%, 47%, 28%, and 38% execution reduction on average for each layer of multiple GNN datasets when compared to baselines, respectively. The mapping and partitioning decisions are estimated by considering both the input graph and the hardware characteristics of the accelerator. All necessary information for this process is either pre-determined or becomes available upon loading the metadata. The execution of the mapping and partitioning algorithms typically requires approximately 100 cycles to reach completion. The latency consumption of each reconfiguration progress for our proposed accelerator (32×32 PE array) is 63 cycles ($2\times32-1$). We should note that the reconfiguration, mapping, and partition progress can be overlapped by the GNN computation in our proposed design. The computation time of all the designs is very close because the amount of multiplication and accumulation computations (MACs) of each layer is the same despite being performed in different architectures.

The proposed design uses routers to ensure the communication between PEs but HyGCN, AWB-GCN, GCNAX, ReGNN, and FLowGNN only use simple interconnects (crossbar, switches, and mesh links) to enable the communication between PEs. The proposed design has reduced the average hop count by using reconfigurable links and routers. The proposed accelerator further alleviates communication contention by implementing a degree-aware mapping algorithm. Notably, the proposed accelerator places lower demands on off-package memory bandwidth when compared to other baseline models as discussed. GCNAX can reduce DRAM access by supporting multiple tiling strategies. FlowGNN and ReGNN can increase performance by supporting multiple levels of parallelism. However, they still suffer from bandwidth limitations, on-chip communication contention, and high communication hop counts, which can negatively impact performance. HyGCN's performance significantly deviates from other baselines due to its deployment of heterogeneous compute engines. The disjoint compute engines result in communication overheads between the aggregation and update phases. Moreover, the inflexible PE architecture suffers from resource under-utilization in the presence of varying vertex and edge quantities. Additionally, data reuse opportunities are not fully exploited in HyGCN.

The main reason our design outperforms other accelerators is that our design has reduced memory bandwidth requirement, reduced DRAM accesses, reduced communication contention, reduced communication hop count, and fully utilized hardware resources which dominate the overall execution time. The hashing-based mapping strategy and sequential execution flow of the compared designs limit their ability to exploit intra-layer data reuse and hardware resources to minimize DRAM

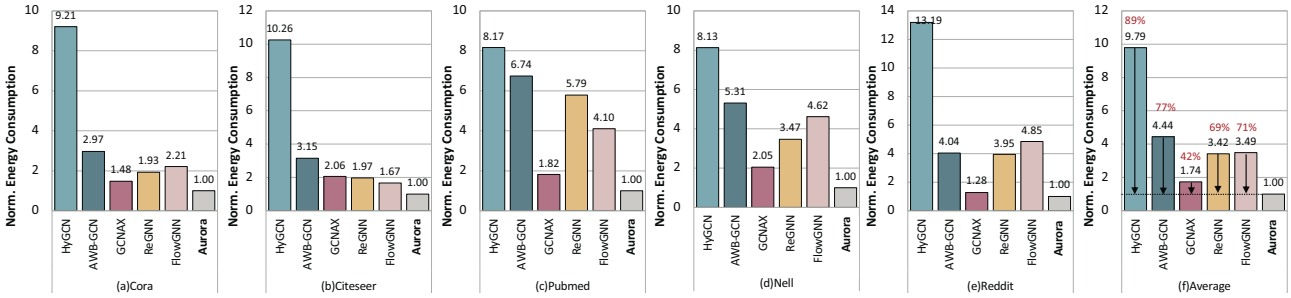

Fig. 10. Normalized energy consumption for each dataset for baselines and our proposed accelerator, normalized to the energy consumption of the proposed accelerator.

accesses. The number of DRAM accesses has a strong impact on performance since it might be the system bottleneck. Our proposed reconfigurable architecture also plays an important role in providing adequate communication patterns and operation options to support various GNN models. ReGNN and FlowGNN suffer from the fixed PE with limited operations or fixed interconnect fabrics because of irregular communication and computation.

As for the speedup for specific datasets, the proposed design performs 5.0–37.0× better over HyGCN, 1.6–3.0× better over AWB-GCN, 1.3–1.9× better over GCNAX, 1.1-2.4× better over ReGNN and 1.1–1.7× better over FlowGNN. The performance gain on the Reddit dataset is not so significant because the density of feature vectors in Reddit (larger than 50%) is higher than that of other datasets, which hinders the performance gains of the proposed design because it will generate more complicated on-chip communication even though the input matrix is not that sparse. Besides, the size of the Reddit dataset is large, which makes it more difficult to exploit the benefits of the degree-aware mapping algorithm.

E. Energy Consumption Analysis

For the energy analysis, it is important to emphasize that the evaluation encompasses the energy consumption of the entire system, encompassing control units, computation units, DRAM, distributed buffers, local buffers, and interconnects. Fig. 10 provides an overview of the normalized overall energy consumption analysis of the proposed accelerator. As can be seen, the proposed accelerator achieves 89%, 77%, 42%, 69%, and 71% energy consumption reduction on average for each dataset by baselines and our proposed accelerator, normalized to the energy consumption of the proposed accelerator. The primary drivers behind these energy savings can be attributed to several factors, including reduced DRAM accesses and diminished on-chip communication latency. These improvements are achieved through a combination of strategies, such as reduced long-distance communication facilitated by the proposed partition algorithm, reduced average hop count thanks to the reconfigurable NoC, and alleviated communication contention through the degree-aware mapping algorithm. The energy consumption of reconfiguration is less than 3% of the overall energy consumption. The energy consumption of reconfiguration is negligible when compared to the overall energy consumption.

F. Area Analysis

We evaluate the area consumption of the various architectures under TSMC 40 nm technology, the MAC array consumes only 7.1% of the total PE area, while the memory hierarchy, SMB, and IDMB/ODMB, consume a majority of the total area, 82.9%. The additional PE control unit and reconfigurable switches consume 3.7% of the total PE area. The total area consumption takes PE, SRAM, flexible interconnects, flexible routers, and control logic into account. For the entire proposed accelerator, the PE array, which consists of 1024 PEs consumes 62.74% of the overall chip area. The controller consumes 0.9% of the total chip area, which is negligible.

The additional components for the flexible interconnect, including flexible routers, horizontal reconfigurable links, reconfigurable links, switches, and muxes consume 5.2% of the total chip area. This shows that the additional modules incur a negligible area overhead for higher performance and energy efficiency.

VII. Conclusion

In this paper, we propose a versatile GNN accelerator that can efficiently facilitate the execution of various GNN models. The proposed architecture can be dynamically configured to multiple sub-accelerators optimized for different GNN execution phases. To achieve this, the proposed accelerator consists of four salient designs - a unified PE architecture, a flexible NoC, a degree-aware mapping, and a partitioning heuristic. The proposed flexible processing unit can be configured to support various computations, including vector-vector multiplication, vector-matrix multiplication, and scalar operations. The proposed NoC design can be dynamically configured to align with varying graph degrees and connectivity by providing additional injection/ejection bandwidth and bridging long-distance communications. The degree-aware mapping is proposed to avoid the unbalanced communication problem caused by high-degree vertices at the aggregation phase. The partitioning approach could efficiently allocate the computing resources to different GNN execution phrases. Consequently, the proposed design can increase hardware utilization with much-improved performance and energy efficiency. Our experimental evaluation clearly demonstrates the substantial benefits of our approach compared to state-of-the-art solutions.

REFERENCES

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

[2] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *Proceeding of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936, 2020.

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

[4] Cen Chen, Kenli Li, Yangfan Li, and Xiaofeng Zou. Regnn: A redundancy-eliminated graph neural networks accelerator. In *Proceeding of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 429–443, 2022.

[5] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference. In *Proceeding of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1099–1112, 2023.

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

[7] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. Machine learning on graphs: A model and comprehensive taxonomy. *The Journal of Machine Learning Research*, 23(1):3840–3903, 2022.

[8] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceeding of the ACM International Conference on World Wide Web (WWW)*, pages 591–600, 2010.

[9] Alberto Garcia Duran and Mathias Niepert. Learning graph representations with embedding propagation. *Advances in neural information processing systems*, 30, 2017.

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

[11] Lei Wang, Yuchun Huang, Yaolin Hou, Shenman Zhang, and Jie Shan. Graph attention convolution for point cloud semantic segmentation. In *Proceeding of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10296–10305, 2019.

[12] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceeding of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 974–983, 2018.

[13] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *Proceeding of the ACM International Conference on World Wide Web (WWW)*, pages 417–426, 2019.

[14] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. T-gcn: A temporal graph convolutional network for traffic prediction. *IEEE transactions on intelligent transportation systems*, 21(9):3848–3858, 2019.

[15] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *Proceeding of the IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10, 2012.

[16] Hang Liu and H Howie Huang. Enterprise: Breadth-first graph traversal on gpus. In *Proceeding of the IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2015.

[17] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)*, 8(1):1–70, 2021.

[18] Sanjay Gandham, Lingxiang Yin, Hao Zheng, and Mingjie Lin. Saga: Sparsity-agnostic graph convolutional network acceleration with near-optimal workload balance. In *Proceeding of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.

[19] Jiaqi Yang, Hao Zheng, and Ahmed Louri. Adapt-flow: A flexible dnn accelerator architecture for heterogeneous dataflow implementation. In *Proceedings of the Great Lakes Symposium on VLSI 2022*, pages 287–292, 2022.

[20] Jiaqi Yang, Hao Zheng, and Ahmed Louri. Venus: A versatile deep neural network accel-erator architecture design for multiple applications. In *Proceedings of Design Automation Conference (DAC)*, 2023.

[21] Jiaqi Yang, Hao Zheng, and Ahmed Louri. Versa-dnn: A versatile architecture enabling high-performance and energy-efficient multi-dnn acceleration. *IEEE Transactions on Parallel and Distributed Systems*, 35(2):349–361, 2024.

[22] Stefanie Jegelka. Theory of graph neural networks: Representation and learning. *arXiv preprint arXiv:2204.07697*, 2022.

[23] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[24] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

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

[26] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *ArXiv*, abs/1810.00826, 2018.

[27] 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*, 30, 2017.

[28] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. Attention-based graph neural network for semi-supervised learning. *arXiv preprint arXiv:1803.03735*, 2018.

[29] Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

[30] Chongyang Zhong, Lei Hu, Zihao Zhang, Yongjing Ye, and Shihong Xia. Spatio-temporal gating-adjacency gcn for human motion prediction. In *Proceeding of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6447–6456, 2022.

[31] Yao Ding, Xiaofeng Zhao, Zhili Zhang, Wei Cai, and Nengjun Yang. Multiscale graph sample and aggregate network with context-aware learning for hyperspectral image classification. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 14:4561–4572, 2021.

[32] Niels Gleinig, Maciej Besta, and Torsten Hoefler. I/o-optimal cache-oblivious sparse matrix-sparse matrix multiplication. In *Proceeding of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 36–46, 2022.

[33] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic graph cnn for learning on point clouds. *ACM Transactions on Graphics (tog)*, 38(5):1–12, 2019.

[34] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceeding of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 161–170, 2015.

[35] Ivica Martinjak and Marin Golub. Comparison of heuristic algorithms for the n-queen problem. In *Proceeding of the IEEE International Conference on Information Technology Interfaces*, pages 759–764, 2007.

[36] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE computer architecture letters*, 10(1):16–19, 2011.

[37] Mark Horowitz. Energy table for 45nm process. In *Stanford VLSI wiki*. 2014.

[38] S Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. Cgra-me: A unified framework for cgra modelling and exploration. In *Proceeding of the IEEE International Conference on Application-specific systems, Architectures and Processors (ASAP)*, pages 184–189, 2017.

[39] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. Polybench: The first benchmark for polystores. In *Proceeding of Springer Performance Evaluation and Benchmarking for the Era of Artificial Intelligence: 10th TPC Technology Conference, TPCTC 2018, Rio de Janeiro, Brazil, August 27–31, 2018, Revised Selected Papers 10*, pages 24–41, 2019.

[40] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[41] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam Hruschka, and Tom Mitchell. Toward an architecture for never-ending language learning. In *Proceeding of the AAAI conference on artificial intelligence*, pages 1306–1313, 2010.