GraphAGILE: An FPGA-Based Overlay Accelerator for Low-Latency GNN Inference

Bingyi Zhang, Hanqing Zeng, and Viktor K. Prasanna, *Fellow, IEEE*

***Abstract*—This article presents GraphAGILE, a domain-specific FPGA-based overlay accelerator for graph neural network (GNN) inference. GraphAGILE consists of (1) *a novel unified architecture design* with an *instruction set*, and (2) *a compiler* built upon the instruction set that can quickly generate optimized code. Due to the proposed instruction set architecture (ISA) and the compiler, GraphAGILE does not require any FPGA reconfiguration when performing inference on various GNN models and input graphs. For the architecture design, we propose a novel hardware module named Adaptive Computation Kernel (ACK), that can execute various computation kernels of GNNs, including general matrix multiplication (GEMM), sparse-dense matrix multiplication (SpDMM), and sampled dense-dense matrix multiplication (SDDMM). The compiler takes the specifications of a GNN model and the graph meta data (e.g., the number of vertices and edges) as input, and generates a sequence of instructions for inference execution. We develop the following compiler optimizations to reduce inference latency: 1) computation order optimization that automatically reorders the computation graph to reduce the total computation complexity, 2) layer fusion that merges adjacent layers to reduce data communication volume, 3) data partitioning with a partition-centric execution scheme that partitions the input graph to fit the available on-chip memory of FPGA, 4) kernel mapping that automatically selects execution mode for ACK, and performs task scheduling to overlap computation with data communication and achieves dynamic load balance. We implement GraphAGILE on a state-of-the-art FPGA platform, Xilinx Alveo U250. GraphAGILE can execute widely used GNN models, including GCN, GAT, GIN, GraphSAGE, SGC and other GNN models supported by GraphGym. Experimental results show that GraphAGILE achieves up to $47.1\times$ ($3.9\times$) reduction in end-to-end latency, including the latency of compilation and hardware execution, compared with the state-of-the-art implementations on CPU (GPU), and achieves up to $2.9\times$ reduction in hardware execution latency compared with the state-of-the-art FPGA accelerators.**

***Index Terms*—Graph neural network, FPGA overlay accelerator, hardware architecture, low-latency inference.**

I. Introduction

GRAPH neural networks (GNNs) have achieved unprecedented success in graph-based machine learning. Compared with traditional algorithms, GNNs achieve superior performance for a wide variety of applications [1], such as recommendation systems, social media [2], etc. *Low-latency GNN inference* is needed in many real-world applications. Examples include real-time traffic prediction [3], and GNN-based scientific simulation [4].

Accelerating GNN inference is challenging because GNN inference [5], [6], [7] requires both sparse and dense computation kernels. While the sparse computation kernels result in poor data reuse and irregular memory access patterns, the dense computation kernels can be executed with regular memory access patterns. General purpose processors (e.g., CPU, GPGPU) are inefficient for GNN inference due to (1) complex cache hierarchy that results in ineffective on-chip memory utilization due to the poor spatial and temporal locality, (2) the general microarchitecture designs are inefficient for various computation kernels in GNNs (i.e., GEMM, SpDMM, and SDDMM). For GPUs, the state-of-the-art GNN frameworks (e.g., Pytorch Geometric (PyG) [8], Deep Graph Library (DGL) [9]) have large inference latency due to (1) large GPU kernel launch time, and (2) sub-optimal execution paradigm for sparse computation leading to large memory traffic. For example, due to the large GPU global memory footprint for storing the intermediate results, programs written with PyG spend 55%–99% [5] time executing the sparse computations of GNN inference.

Many GNN accelerators [5], [6], [7], [10], [11], [12], [13], [14], [15] have been proposed to overcome the inefficiency of CPUs and GPUs. Previous works either directly design accelerators for specific GNN models [10], [11] or develop design automation frameworks [6], [12], [13] to generate FPGA accelerators for a specific GNN model and an input graph. However, the design automation frameworks need to regenerate optimized hardware design if the structure of the GNN model or the topology of the input graph changes. The hardware regeneration requires meta compilation, hardware synthesis, place&route, and FPGA reconfiguration, which incur significant overhead and are not suitable for cloud-based FPGA accelerators. A typical end user may explore a variety of GNN models and perform inference on various input graphs. Moreover, in a cloud-based system, multiple users share the same FPGA. Different users may run different GNN models with different input graphs. Therefore, the time-consuming process of regenerating an optimized accelerator makes the design automation frameworks unattractive in the above scenarios.

In this paper, we propose an FPGA-based overlay accelerator, GraphAGILE. An FPGA overlay [16], [17] consists of

Manuscript received 19 September 2022; revised 28 March 2023; accepted 15 June 2023. Date of publication 20 June 2023; date of current version 21 July 2023. The work of Bingyi Zhang and Viktor Prasanna was supported by the National Science Foundation (NSF) under Grants CCF-1919289 and OAC-2209563. Recommended for acceptance by F. Petrini. *(Corresponding author: Bingyi Zhang.)*

Bingyi Zhang and Viktor K. Prasanna are with the Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA 90089 USA (e-mail: bingyizh@usc.edu; prasanna@usc.edu).

Hanqing Zeng is with Meta Platforms, Inc., Menlo Park, CA 94025 USA (e-mail: zengh@meta.com).

Digital Object Identifier 10.1109/TPDS.2023.3287883

an instruction set architecture (ISA) and a compiler, providing software-like programmability and targeting a specific application domain. The ISA of GraphAGILE unifies the execution of both the sparse and dense computation kernels of GNNs without hardware reconfiguration. To program the ISA, the compiler takes as inputs the specification of the GNN model and the graph meta data, and generates a sequence of instructions to execute on the ISA of the overlay architecture. To reduce the inference latency, we propose several optimizations for the compiler to efficiently utilize the ISA. To the best of our knowledge, GraphAGILE is the first FPGA overlay accelerator for GNNs. We summarize our main contributions as follows:

- We propose an instruction set architecture to accelerate GNN inference. It supports a broad range of GNN models by efficiently executing various computation kernels in GNNs, including GEMM, SpDMM, and SDDMM.
- We develop a compiler that generates an instruction sequence based on an input graph and GNN model. Compiler optimizations include:
  - computation order optimization that automatically reorders the computation graph to reduce the total computation complexity.
  - layer fusion that merges adjacent layers to communicate the inter-layer results through on-chip memory, which reduces the total volume of external memory communication.
  - graph partitioning that optimizes the intra-layer and inter-layer data communication under a given on-chip memory constraint.
  - kernel mapping and task scheduling that hide data communication latency and achieve dynamic load balance.
- We deploy GraphAGILE on Xilinx Alveo U250, a state-of-the-art cloud-based FPGA platform.
  - We demonstrate that GraphAGILE can execute widely used GNN models, including GCN, GAT, GIN, GraphSAGE, SGC, and other GNN models supported by GraphGym [18].
  - GraphAGILE achieves up to $47.1\times$ ($3.9\times$) speedup in end-to-end latency (see Section VIII) compared with the state-of-the-art implementations on CPU (GPU), and up to $2.9\times$ speedup in hardware execution latency compared with the state-of-the-art FPGA accelerators.

The rest of the paper is organized as follows: Section II introduces the background of graph neural networks; Section III covers the related work; Section IV presents an overview of GraphAGILE; Section V describes the microarchitecture design of GraphAGILE; Section VI covers the details of the compiler design; Section VII describes the implementation details and Section VIII includes the evaluation results.

II. Background

This section introduces the background of graph neural networks and briefly describes two well-known graph neural network models.

A. Graph Neural Networks

Table I defines the notations in GNN layer operations. Graph neural networks (GNNs) [19], [20] are proposed for representation learning on a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Each edge in $\mathcal{G}$ is associated with a weight. A GNN model consists of a stack of GNN layers. Each GNN layer performs message passing on $\mathcal{G}$ where each vertex aggregates information from its neighbors. Thus, a multi-layer GNN model recursively performs such message passing on multi-hop neighbors. According to [8], [9], a GNN layer can be abstracted as

TABLE I
Notations

Notation	Description	Notation	Description
$\mathcal{G}(\mathcal{V}, \mathcal{E})$	input graph	v_i	i^{th} vertex
$\mathcal{V}$	set of vertices	$e(i,j)$	edge from v_i to v_j
$\mathcal{E}$	set of edges	L	number of GNN layers
$\boldsymbol{h}_i^l$	feature vector of v_i at layer l	$\boldsymbol{m}_i^l$	aggregated message by vertex v_i

$$\text{Edge-wise}: \boldsymbol{m}_e^l = \phi(\boldsymbol{h}_u^{l-1}, \boldsymbol{h}_v^{l-1}, w_e^{l-1}), \forall e(u,v) \in \mathcal{E} \quad (1)$$

$$\text{Node-wise}: \boldsymbol{h}_v^l = \psi(\boldsymbol{h}_v^{l-1}, \rho(\{\boldsymbol{m}_e^l : e(u,v) \in \mathcal{E}\})), \quad (2)$$

where $\phi()$ is the *message function*. Each edge uses $\phi()$ to generate a message by combining the edge weight w_e^{l-1} with the features of its incident vertices. $\psi()$ is the *update function*. Each vertex uses $\psi()$ to update its features by aggregating the incoming messages using the *reduction function* $\rho()$. In GNNs, the message/update functions are parameterized by neural network modules [20], such as Multi-layer Perception. Some well-known GNN models include:

GCN [19]: each layer is defined as

$$\boldsymbol{m}_i^l = \text{Sum}\left(\left\{\alpha_{ji} \cdot \boldsymbol{h}_j^{l-1} : j \in \mathcal{N}(i) \cup \{i\}\right\}\right)$$
$$\boldsymbol{h}_i^l = \text{ReLU}\left(\boldsymbol{m}_i^l \boldsymbol{W}^l\right), \quad (3)$$

where l denotes the l^{th} layer, $\alpha_{ji} = \frac{1}{\sqrt{D(j) \cdot D(i)}}$ ($D(j)$ is the degree of v_j), $\boldsymbol{W}^l$ denotes the weight matrix of layer l, and $\mathcal{N}(i)$ denotes the set of neighbors of v_i.

GAT [21]: this model has similar layer definition as GCN. In addition, GAT applies the attention mechanism to calculate edge weight α_{ij} dynamically

$$\alpha_{ij} = \frac{\exp\left(\text{LReLU}\left(\langle \boldsymbol{a}_{\text{att}}, [\boldsymbol{W}_{\text{att}}\boldsymbol{h}_i || \boldsymbol{W}_{\text{att}}\boldsymbol{h}_j] \rangle\right)\right)}{\sum_{k \in \mathcal{N}(i)} \exp\left(\text{LReLU}\left(\langle \boldsymbol{a}_{\text{att}}, [\boldsymbol{W}_{\text{att}}\boldsymbol{h}_i || \boldsymbol{W}_{\text{att}}\boldsymbol{h}_k] \rangle\right)\right)}, \quad (4)$$

where $\boldsymbol{a}_{\text{att}}$ is an attention vector, $\boldsymbol{W}_{\text{att}}$ is an attention matrix, and $\langle , \rangle$ is the vector inner product operator.

In addition, many other GNN models (e.g., GIN [22]) have been proposed following the recursive message-passing paradigm. Recently, [18] proposes the GraphGym library [18] and defines the general design space of a GNN. The design space includes intra-layer design and inter-layer design, where the intra-layer design follows the message-passing paradigm defined in (1) and (2), the inter-layer design adds the residual connections across the GNN layers.

III. Related Work

There have been many FPGA overlay accelerators proposed for CNNs targeting image-related tasks, such as AMD Xilinx DPU [23], Intel DLA [17], Nvidia NVDLA [24], TVM

VTA [25], and OPU [16]. These CNN overlay accelerators have similar components: 1) a general architecture design with an instruction set, and 2) a compiler that generates the instruction sequence for the target CNN model. Compared with CNN overlay accelerators, the design of GNN overlay accelerators is more challenging: 1) The major computation kernel in CNNs is convolution, which can be efficiently supported by a single hardware design, such as a 2-D systolic array. In contrast, GNNs have heterogeneous computation kernels (e.g., GEMM, SpDMM, and SDDMM), making it more challenging to design a unified hardware architecture. 2) For a CNN overlay accelerator, the compiler processes images of regular shapes. For tasks such as image classification, CNN models accept input images of a fixed size, and the compiler only needs to generate a single instruction sequence for these models. In contrast, the input graphs to GNNs are independent of GNN models, and the real-world graphs have various sizes and connectivity. The graphs with the same number of vertices and edges may have highly different structures. Thus, the software compiler of the GNN overlay accelerator needs to process graphs of various sizes and connectivity. Complex data-dependent optimizations (e.g., complex graph partitioning) in compiler result in large overhead at compilation time, which may degrade the end-to-end latency (see Section VIII).

In our prior work [10], we proposed a hybrid hardware architecture to accelerate graph convolutional network (GCN) [19]. In [6], we proposed a partition-centric execution scheme to accelerate GCN by improving the memory performance. In [26], we proposed a unified hardware architecture that supports GEMM and SpDMM in GNNs. Based on our previous works [6], [10], [26], we propose 1) a unified hardware architecture to support three key computation kernels in GNNs, including GEMM, SpDMM, and SDDMM, 2) a general instruction set that enables software-like programmability and supports a broad range of GNNs, and 3) a compiler with latency reduction optimizations (including the computation order optimization in [10], and the partition-centric execution scheme in [6]) to automatically compile the GNN model into an instruction sequence for hardware execution.

IV. Overview

In this section, we introduce the GraphAGILE workflow (Section IV-A) and provide an overview of the hardware architecture (Section IV-B).

A. Overview of GraphAGILE

Target Application Domain: This work targets the inference process of various GNN-based applications, such as recommendation system [20], social media, citation networks [19], etc. In the target applications, the graphs can be very large. For example, a graph in recommendation systems may contain billions of vertices and edges. GraphAGILE supports a broad range of GNN models, including (1) widely used GNN models (GCN [19], GraphSAGE [20], GAT [21], GIN [22], SGC [27]), (2) GNN models in the design space of GraphGym [18]. In addition, GraphAGILE has the potential to be applied to other GNN models. An *instance* to GraphAGILE is specified by (1) the specifications of a GNN model, (2) the specifications of an input graph.

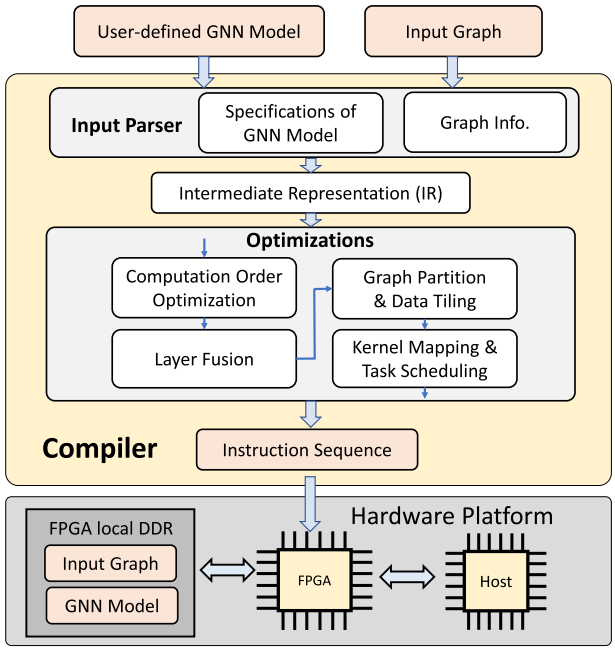

Fig. 1. Overview of GraphAGILE.

Hardware Platform: The hardware platform consists of an FPGA device, FPGA local DDR memory, and a host processor. The proposed hardware accelerator is deployed on the FPGA device. FPGA local DDR memory stores the input graph, the GNN model, and binary files generated by the compiler. The compiler is executed on the host processor.

Compiler: Users define the GNN using Pytorch Geometric (PyG) library. The inputs to the compiler are (1) the computation graph of the GNN model generated by PyG, and (2) the input graph. The Input Parser (Fig. 1) extracts the specifications of the GNN model and the information of the input graph to generate the *Intermediate Representation (IR)*. After obtaining IR, the compiler performs the four optimization steps on the GNN computation graph as shown in Fig. 1. Then, the compiler generates a sequence of instructions to execute on the hardware accelerator.

B. Architecture Overview

Fig. 2 depicts the proposed hardware architecture. There are N_{pe} Processing Elements (PEs) working in parallel. At runtime, the Scheduler reads the executable/binary file from the FPGA DDR and assigns the workload to PEs (see Section VI-F). Each PE has an Instruction Queue (IQ) to receive the incoming instructions assigned by the Scheduler. The Instruction Decoder & Control Signal Generator reads the instructions from IQ and generates the control signals for the hardware modules. Each PE has a Weight Buffer to store the weight matrices, an Edge Buffer to store the edges, and a Feature Buffer to store the vertex feature vectors. Each buffer has a data loader&writer that communicates with the FPGA DDR. Each PE has an Adaptive Computation

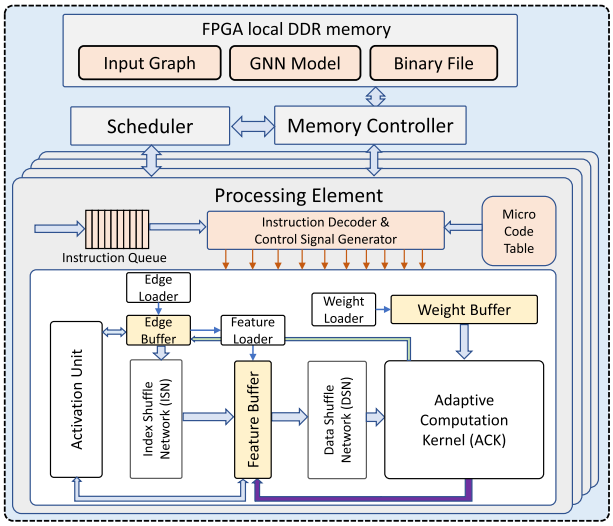

Fig. 2. Hardware architecture.

Kernel, which is the key novelty in our hardware design. The Adaptive Computation Kernel (ACK, Fig. 5) can execute various computation kernels of GNNs.

Hardware Parameters: The proposed architecture is defined by the following hardware parameters: (1) the number of Processing Elements N_{pe}, (2) the dimension of the Adaptive Computation Kernel (ACK) $P_{\text{sys}} \times P_{\text{sys}}$, (3) dimensions of buffers, including the dimension of Weight Buffer $N_W \times P_{\text{sys}}$, the dimension of Edge Buffer $N_E \times 3$, the dimension of Feature Buffer $N_{F1} \times N_{F2}$, (4) the set of arithmetic operations supported by the ACK and the Activation Unit.

V. Microarchitecture

In this section, we first introduce the data format used by GraphAGILE (Section V-A) and then discuss the key computation kernels in GNNs (Section V-B). Next, we describe the proposed instruction set of GraphAGILE (Section V-C), and provide details of the microarchitecture that supports the proposed instruction set, including the datapath (Section V-D) and the on-chip memory organization (Section V-E).

A. Graph Data Format

We use $\boldsymbol{h}_i^l$ to denote the *feature vector* of vertex v_i at layer l (Table I). We use the Coordinate Format (COO) to capture all graph *edges*. Each edge is a 3-tuple $(src, dst, weight)$ denoting the source vertex index, destination vertex index, and edge weight, respectively. We construct the feature matrix $\boldsymbol{H}$ by stacking feature vectors. Each row of $\boldsymbol{H}$ is the feature vector of a vertex. Denote $\boldsymbol{A}$ as the sparse adjacency matrix where for an edge (u, v, w), we have $\boldsymbol{A}_{u,v} = w$.

B. Computation Kernels in GNNs

We identify the following key computation kernels:

General Dense-Dense Matrix Multiplication (GEMM): $\phi()$, $\psi()$ and $\rho()$ can involve GEMM, where the feature matrix $\boldsymbol{H}$ is multiplied by weight matrix $\boldsymbol{W}$. For example, in the $\phi()$ of GAT [21], $\boldsymbol{H}$ is multiplied by $\boldsymbol{W}_{\text{att}}$ for calculating edge weights ((4)). In $\psi()$ and $\rho()$ of GraphSAGE [20], $\boldsymbol{H}$ is multiplied by $\boldsymbol{W}$ to obtain the updated feature vectors. In general, $\boldsymbol{H}$ is a large dense matrix with height equal to $|\mathcal{V}|$ and $\boldsymbol{W}$ is a small dense matrix (e.g., $\boldsymbol{W}$ has size 256×256 in [20]).

Sparse-Dense Matrix Multiplication (SpDMM): According to (1) and (2), the vertices propagate the messages $\boldsymbol{m}_e^{l+1}$ along the outgoing edges. Then each vertex aggregates the incoming messages through $\rho()$. The above message passing process is equivalent to SpDMM $\boldsymbol{A} \cdot \boldsymbol{H}$.

Sampled Dense-Dense Matrix Multiplication (SDDMM): According to [8], in edge-wise computation ((1)), many GNN models calculate edge weights using the dot product of the feature vectors of the source and destination vertices. The above computation process corresponds to SDDMM operation $\boldsymbol{A} \odot (\boldsymbol{H}\boldsymbol{H}^T)$, where $\odot$ is the element-wise multiplication. *Sampled* means that the required results are sampled from $(\boldsymbol{H}\boldsymbol{H}^T)$ based on the non-zero elements in $\boldsymbol{A}$. For each non-zero element $\boldsymbol{A}_{i,j}$. we calculate $\boldsymbol{A}_{i,j} = \langle \boldsymbol{H}_i, \boldsymbol{H}_j \rangle$. Therefore, the basic operation in SDDMM is the vector inner product.

Other Computation Kernels: GNNs also involve vector addition (e.g., residual connection), element-wise activation (e.g., ReLU, Softmax), batch normalization.

C. Instruction Set

1) High-Level Instructions: The proposed instruction set is comprised of *high-level instructions* and *microcode*. All the high-level instructions have a uniform 128-bit length, and the instruction fields are depicted in Fig. 3. The `OPCODE` field indicates the type of instruction. Other fields contain instruction-specific information.

- *Control and Scheduling Instruction (CSI):* A CSI contains the meta data of a computation layer in the intermediate representation (Section VI-A). Based on the CSI, the scheduler assigns the workloads of a layer to the PEs.
- *Memory Read/Write Instruction:* A memory read/write instruction initiates data communication (model weights, edges, vertex feature vectors) with FPGA DDR memory.
- *GEMM Instruction:* A GEMM instruction contains the information (e.g., matrix size, buffer ID that stores the matrices) of the matrix multiplication between the weight matrix (in the Weight Buffer) and feature matrix (in the Feature Buffer).
- *SpDMM Instruction:* A SpDMM instruction performs multiplication of $\boldsymbol{A}$ and $\boldsymbol{H}$. The instruction specifies the number of non-zero elements in $\boldsymbol{A}$ (which enables edge-centric computation of SpDMM. See Section V-D) and buffer ID that stores $\boldsymbol{A}$.
- *SDDMM Instruction:* Similar to the SpMM instruction, it specifies the number of non-zero elements in $\boldsymbol{A}$ and the buffer IDs that store $\boldsymbol{A}$ and $\boldsymbol{H}$.
- *Other instructions:* There are other instructions including the Initialization Instruction, Activation Instruction, etc.

2) Microcode: A high-level instruction defines a computation task in coarse-grained granularity. To execute a high-level instruction, the Instruction Decoder & Control Signal Generator

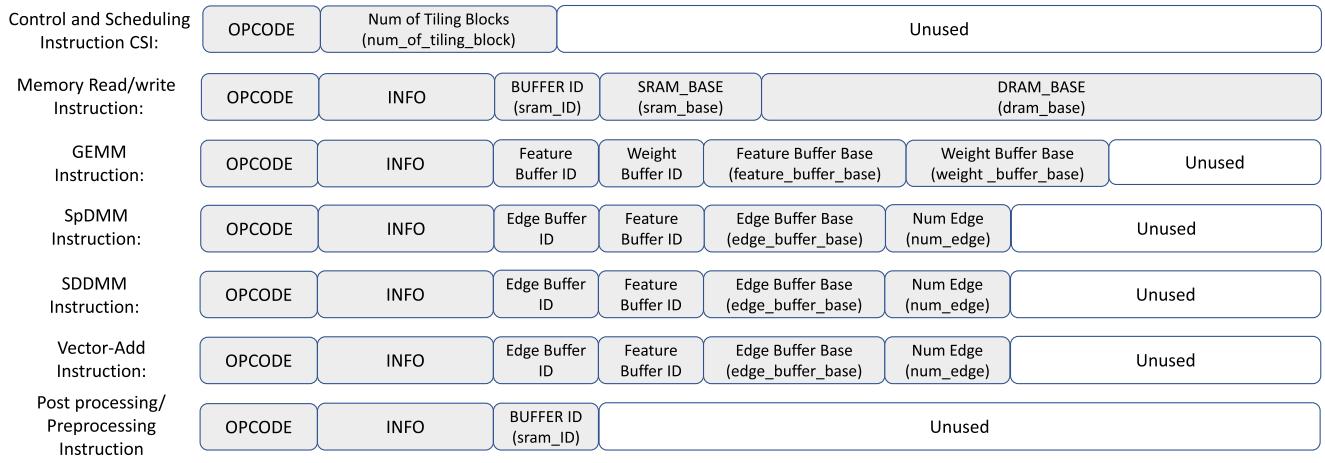

Fig. 3. GraphAGILE high-level instruction fields.

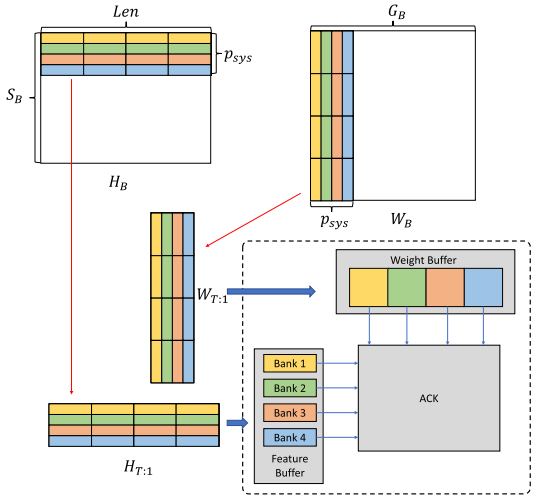

Fig. 4. GEMM between a block of feature matrix $\boldsymbol{H}_B$ (stored in Feature Buffer) and a block of weight matrix $\boldsymbol{W}_B$ (stored in Weight Buffer).

translates it to a sequence of microcode that has fine-grained granularity that can be executed by ACK. The translation is through looking up the Microcode Table. For example, a GEMM instruction defines the multiplication of a large feature matrix (stored in Feature Buffer) and a large weight matrix (stored in Weight Buffer). The GEMM instruction is decomposed into block matrix multiplication (BlockMM), where block size corresponds to the dimension of ACK. The microcodes of GEMM use a three-level nested loop to execute the BlockMM on ACK. The microcodes of GEMM, SpDMM, and SDDMM are described as follows:

Microcode of GEMM Instructions: A high-level GEMM instruction is translated to a sequence of microcode to execute the GEMM between a block of feature matrix $\boldsymbol{H}_B \in \mathbb{R}^{S_B \times Len}$ and a block of weight matrix $\boldsymbol{W}_B \in \mathbb{R}^{Len \times G_B}$. The computation process of GEMM is illustrated in Fig. 4. The Pseudocode of the sequence of microcode is described in Algorithm 1. In GEMM mode, the Adaptive Computation Kernel (ACK) works as a 2-D systolic array of size $p_{\text{sys}} \times p_{\text{sys}}$ using output-stationary

Algorithm 1: Pseudocode of GEMM Microcode.

Input: $\boldsymbol{H}_B$; $\boldsymbol{W}_B$
Output: $\boldsymbol{H}_{\text{out}}$
1: **for** $i \leftarrow 1$ to $\frac{S_B}{p_{\text{sys}}}$ **do**
2: **for** $j \leftarrow 1$ to $\frac{G_B}{p_{\text{sys}}}$ **do**
3: // Pipelined execution of $\boldsymbol{H}_{\text{out}:ij} = \boldsymbol{H}_{T:i} \times \boldsymbol{W}_{T:j}$
4: **for** $k \leftarrow 1$ to Len **Parallel do**
5: Load the p_{sys} data of k^{th} column of $\boldsymbol{H}_{T:i}$ and send them to ACK
6: Load the p_{sys} data of k^{th} row of $\boldsymbol{W}_{T:j}$ and send them to ACK

Algorithm 2: Pseudocode of SpDMM Microcode.

Input: $\boldsymbol{H}_B$; $\boldsymbol{A}_B$; number of edges in $\boldsymbol{A}_B$: N_e
Output: $\boldsymbol{H}_{\text{out}}$
1: **for** $i \leftarrow 1$ to $\frac{2N_e}{p_{\text{sys}}}$ **do** ▷Pipelined execution of SpDMM
2: Load $\frac{p_{\text{sys}}}{2}$ unprocessed edges from $\boldsymbol{A}_B$ in Edge Buffer
3: Send the $\frac{p_{\text{sys}}}{2}$ edges to Index Shuffle Network (ISN)

dataflow. In each clock cycle, ACK receives p_{sys} data from Feature Buffer and p_{sys} data from Weight Buffer, respectively. In Feature Buffer, $\boldsymbol{H}_B$ is further partitioned to small data tiles along row dimensions, and each data tile $\boldsymbol{H}_{T:i}$ has p_{sys} rows. Similarly, in Weight Buffer, $\boldsymbol{W}_B$ is partitioned to small data tiles along column dimension and each data tile $\boldsymbol{W}_{T:i}$ has p_{sys} columns of $\boldsymbol{W}_B$. $\boldsymbol{H}_{\text{out}:ij}$ denotes the result of the multiplication between $\boldsymbol{H}_{T:i}$ and $\boldsymbol{W}_{T:j}$.

Microcode of SpDMM Instructions: A high-level SpDMM instruction is translated to a sequence of microcode to execute the SpDMM between a block of feature matrix $\boldsymbol{H}_B$ (stored in the Feature Buffer) and a block of sparse adjacency matrix $\boldsymbol{A}_B$ (stored in the Edge Buffer). The execution of SpDMM is edge-centric (See Section V-D). Therefore, in each clock cycle, $\frac{p_{\text{sys}}}{2}$ unprocessed edges in $\boldsymbol{A}_B$ are fetched from Edge Buffer. The $\frac{p_{\text{sys}}}{2}$ edges are sent to Index Shuffle Network to execute feature aggregation.

Algorithm 3: Pseudocode of SDDMM Microcode.

Input: $\boldsymbol{H}_B$; $\boldsymbol{A}_B$; number of edges in $\boldsymbol{A}_B$: N_e
Output: weights of all the edges in $\boldsymbol{A}_B$
1: **for** $i \leftarrow 1$ to $\frac{2N_e}{p_{sys}}$ **do** ▷Pipelined execution of SDDMM
2: Load $\frac{p_{sys}}{2}$ unprocessed edges from $\boldsymbol{A}_B$ in Edge Buffer
3: Extract the $\frac{p_{sys}}{2}$ src indices and $\frac{p_{sys}}{2}$ dst indices
4: Send the p_{sys} indices to ISN

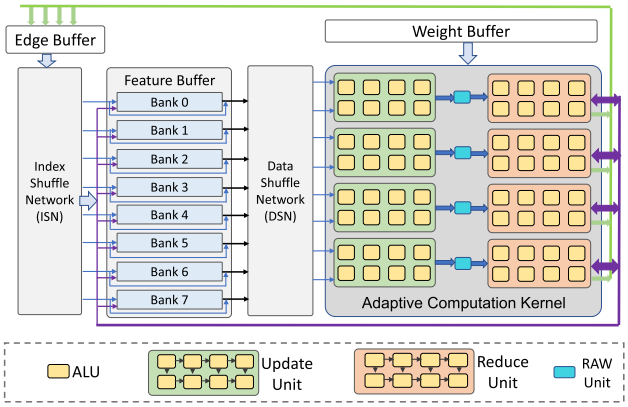

Fig. 5. Adaptive Computation Kernel (when $p_{sys} = 8$) with ISN and DSN. The interconnections among ALUs are specified in Fig. 6.

Microcode of SDDMM Instructions: A high-level SDDMM instruction is translated to a sequence of microcode to execute the SDDMM using a block of feature matrix $\boldsymbol{H}_B$ (stored in the Feature Buffer) and a block of sparse adjacency matrix $\boldsymbol{A}_B$ (stored in the Edge Buffer). Similar to SpDMM, the execution of SDDMM is edge-centric (See Section V-D). In each clock cycle, $\frac{p_{sys}}{2}$ unprocessed edges in $\boldsymbol{A}_B$ are fetched from Edge Buffer. The $\frac{p_{sys}}{2}$ src indices and $\frac{p_{sys}}{2}$ dst indices are extracted from the $\frac{p_{sys}}{2}$ unprocessed edges. Then, the total p_{sys} indices are sent to Index Shuffle Network (ISN) to execute the SDDMM of $\boldsymbol{A}_B$ and $\boldsymbol{H}_B$.

D. Various Execution Modes

As shown in Fig. 5, an ACK contains an array of Arithmetic Logic Units (ALUs) of size $p_{sys} \times p_{sys}$, where p_{sys} is the power of 2. An ALU can execute various arithmetic operations, including Multiplication, Addition, Accumulation, Min, Max, etc. The interconnections among ALUs are shown in Fig. 6. The array of ALUs is divided into Update Units and Reduce Units. An Update Unit or a Reduce Unit has size $(p_{sys}/2) \times 2$. The Feature Buffer has p_{sys} parallel memory banks. $\boldsymbol{h}_i$ is stored in bank $(i \mod p_{sys})$. There are two interconnection networks – Index Shuffle Network (ISN) and Data Shuffle Network (DSN). The ISN routes edges to the memory banks of Feature Buffer for fetching the features of incident vertices. The DSN routes the input data (vertex features with the edge) to Adaptive Computation Kernel. The routing is based on the least significant $\log(p_{sys})$ bits of the vertex index. The ACK has various execution modes, including *GEMM mode*, *SpDMM mode*, *SDDMM mode*, and *Vector-Addition mode*. Each ALU maintains multiplexers with control logic to select the input and output ports for an execution mode. The mode switching incurs the overhead of only one clock cycle.

Algorithm 4: SpDMM Following Scatter-Gather Paradigm.

while not done **do**
for each edge $e(src, dst, weight)$ **do** ▷Scatter Phase
Fetch $src.features$ from Feature Buffer
Form input pair $(src.features, e)$
for each input pair **do** ▷Gather Phase
Produce $u \leftarrow$Update$(src.features, e.weight)$
Update $v_{dst} \leftarrow$ Reduce$(u.features)$

GEMM Mode: The array of ALUs is organized as a two-dimensional systolic array with fully localized interconnection. GEMM mode supports dense matrix multiplication of feature matrix $\boldsymbol{H}$ and weight matrix $\boldsymbol{W}$. Weight Buffer streams the weight matrix into the systolic array, and Feature Buffer streams multiple vertex feature vectors into the systolic array. Systolic array of size $p_{sys} \times p_{sys}$ executes p_{sys}^2 Multiply-Accumulation operations per clock cycle.

SpDMM Mode: As shown in Algorithm 4, SpDMM is executed following the Scatter-Gather paradigm. The array of ALUs in ACK is divided into multiple Update Units and Reduce Units. In each Update Unit, the ALUs are organized as a vector multiplier that multiplies the vertex feature vector by the edge weight. In each Reduce Unit, the ALUs execute the element-wise reduction operation $\rho()$. Suppose a vertex is defined by $(src, features)$, where src denotes the source vertex index and the $features$ is the feature vector of the source vertex. The generated intermediate results by the Update Units are represented by $(dst, features)$. The intermediate results are applied to the destination vertex v_{dst} by the Reduce Unit. An Update Unit and a Reduce Unit form an "UR-pipeline". The computation of SpDMM is driven by unprocessed edges (i.e., *edge-centric processing* [28]). Unprocessed edges are fetched from Edge Buffer to ISN. In ISN, an edge e is routed to the corresponding memory bank in Feature Buffer to fetch $src.features$, thus forming the input pair $(src.features, e)$. Then, the DSN routes the input pairs to the UR pipelines based on the dst of the edge. The input pairs having $e.dst = i \times p_{sys} + k$ $(0 \leqslant k < p_{sys})$ will be routed to the $\lfloor k/2 \rfloor^{\text{th}}$ UR pipeline. This is because the output port of $\lfloor k/2 \rfloor^{\text{th}}$ UR pipeline is connected to bank $\lfloor k/2 \rfloor$ and bank $\lfloor k/2 \rfloor + 1$ of Feature Buffer, where $v_{e.dst}$ is stored. Then, the UR pipeline processes the input pair, and the intermediate result generated by the input pair is applied to the destination vertex $v_{e.dst}$. $p_{sys}/2$ input pairs can be processed by the $p_{sys}/2$ UR pipelines concurrently.

SDDMM Mode: The basic operation is the inner product of two feature vectors. For each edge (src, dst), the feature vectors $\boldsymbol{h}_{src}$ and $\boldsymbol{h}_{dst}$ are fetched from the Feature Buffer. The result of the inner product of $\boldsymbol{h}_{src}$ and $\boldsymbol{h}_{dst}$ becomes the weight of the edge (src, dst). To support the inner-product, the ALUs in a UR pipeline form a multiply-adder tree. The topological structure of the multiply-adder tree is shown in Fig. 6. Similar to SpDMM, the execution of SDDMM is edge-centric. For an edge (src, dst), src and dst are routed to the corresponding

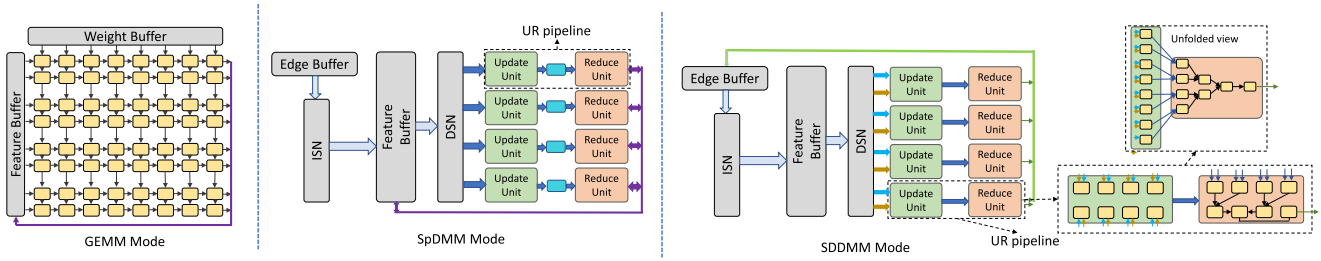

Fig. 6. The datapath of GEMM mode, SpDMM mode, SDDMM mode.

memory banks of Feature Buffer to fetch $\boldsymbol{h}_{src}$ and $\boldsymbol{h}_{dst}$. The inner product of $\boldsymbol{h}_{src}$ and $\boldsymbol{h}_{dst}$ is executed by a UR pipeline. The ACK can execute $p_{\text{sys}}/2$ vector inner products of length p_{sys} during each clock cycle. The dot product of two feature vectors of length $|\boldsymbol{h}_i|$ is executed in $\left\lceil \frac{|\boldsymbol{h}_i|}{p_{\text{sys}}} \right\rceil$ cycles and the intermediate result is stored at the root node of the adder tree for accumulation.

Vector Addition Mode: In Vector Addition Mode, the basic operation is the addition of two feature vectors. An Update Unit (See Fig. 5) works as a vector adder. To add $\boldsymbol{h}_{\text{u}}$ and $\boldsymbol{h}_{\text{v}}$, the indices u and v are routed through Index Shuffle Network (ISN) to Feature Buffer to fetch $\boldsymbol{h}_{\text{u}}$ and $\boldsymbol{h}_{\text{v}}$. Then $\boldsymbol{h}_{\text{u}}$ and $\boldsymbol{h}_{\text{v}}$ are routed to an Update Unit through Data Shuffle Network (DSN) to perform vector addition. The results will bypass the Reduction Unit and are sent back to Feature Buffer. The ACK can execute $p_{\text{sys}}/2$ vector additions of length p_{sys} at each clock cycle. Two feature vectors of length $|\boldsymbol{h}_i|$ can be added in $\lceil \frac{|\boldsymbol{h}_i|}{p_{\text{sys}}} \rceil$ cycles.

E. Parallel On-Chip Memory Access

The Feature Buffer supports parallel memory access patterns of various computation modes enabled by ISN and DSN. Feature Buffer has p_{sys} parallel memory banks, and the feature vector of vertex v_i is stored in bank $(i \mod p_{\text{sys}})$. Edge Buffer can output p_{sys} edges at each clock cycle by having port width $p_{\text{sys}}d_e$, where d_e is the bit width of an edge. ISN performs all-to-all interconnection between Edge Buffer and Feature Buffer. DSN performs all-to-all interconnection between Feature Buffer and ACK. The ISN and the DSN are implemented using the butterfly network [29]

Parallel Memory Accesses in GEMM Mode: The ACK directly fetches p_{sys} features from p_{sys} memory banks per clock cycle. No data shuffling is required for GEMM. The Weight Buffer also has p_{sys} memory banks that can output p_{sys} data of the weight matrix each clock cycle.

Parallel Memory Accesses in SpDMM Mode: $p_{\text{sys}}/2$ edges $\{e_1, e_2, \ldots, e_{p_{\text{sys}}/2}\}$ are sent to ISN simultaneously. The edges are routed to the corresponding memory banks of Feature Buffer based on their src. $p_{\text{sys}}/2$ edges will generate $p_{\text{sys}}/2$ input pairs $(src.features, e)$ after fetching the feature vectors. Then the $p_{\text{sys}}/2$ input pairs are routed to the corresponding UR pipelines based on their $e.dst$.

Parallel Memory Accesses in SDDMM Mode: $p_{\text{sys}}/2$ edges $\{e_1, e_2, \ldots, e_{p_{\text{sys}}/2}\}$ are fetched from the Edge Buffer in each cycle. The $p_{\text{sys}}/2$ src indices and $p_{\text{sys}}/2$ dst indices $\{src_1, dst_1, src_2, dst_2, \ldots, src_{p_{\text{sys}}/2}, dst_{p_{\text{sys}}/2}\}$ of $p_{\text{sys}}/2$ edges are sent to p_{sys} input ports of ISN. The ISN routes the p_{sys} indices to the Feature Buffer to fetch the p_{sys} vertex feature vectors from the Feature Buffer. Then, the p_{sys} feature vectors are routed to the $p_{\text{sys}}/2$ UR pipelines of ACK. The i^{th} UR pipeline performs the inner product of $\boldsymbol{h}_{src_i}$ and $\boldsymbol{h}_{dst_i}$.

VI. Compiler

We develop a compiler that reads the user-defined GNN model and input graph, and generates a sequence of instructions. User defines the GNN model using the high-level API in Pytorch Geometric (PyG) Library [8], which is a general framework for GNNs. There are two phases for instruction generation – *translation phase* and *optimization phase*. In the translation phase, the Input Parser generates the Intermediate Representation (IR) from the inputs. In the optimization phase, the compiler performs four-step optimizations and generates the output instruction sequence: 1) *Step 1*: the compiler reorders the computation graph based on the theoretical computation complexity. 2) *Step 2*: the compiler merges some adjacent layers to communicate intermediate data through on-chip memory. 3) *Step 3*: the compiler performs data partitioning based on the available on-chip memory to optimize off-chip data communication and enable dynamic task scheduling, 4) *Step 4*: the compiler maps various kernels to ACK, and performs task scheduling to hide the data communication overhead and achieve dynamic load balance.

A. Intermediate Representation

We define a unified Intermediate Representation (IR) for each type of computation layer (Table II). A GNN layer can be decomposed into a sequence of computation layers. We identify six types of computation layers – *Aggregate*, *Linear*, *Vector-Inner*, *Vector-Add*, *Activation* and *BatchNorm*. The six types of layers can represent a broad range of models because (1) the key computation kernels of GNNs (SpDMM, GEMM, and SDDMM) can be represented as *Aggregate*, *Linear*, or *Vector-Inner*, (2) the auxiliary kernels such as non-linear activation, residual connection, batch normalization can be represented using others lightweight layers (e.g., *Vector-Add*, *Activation*, and *BatchNorm*). The compiler translates the GNN model to a computation graph, with each node being the IR of a layer. For example, the GNN model [19] in Listing 1 is translated to the computation graph in Fig. 7. The abstraction of each type of computation layer is described in the following:

TABLE II
IR OF A COMPUTATION LAYER

Layer Type	Aggregate(0), Linear(1), Vector-Inner(2), Vector-Add(3), Activation(4), BatchNorm(5)
Layer ID	1,2,3,...
Parent Layer IDs	Parent1_ID, ...
Child Layer IDs	Child1_ID, ...
Input Dimension	f_{in}
Output Dimension	f_{out}
# of vertices	$\lvert\mathcal{V}\rvert$
# of edges	$\lvert\mathcal{E}\rvert$
Aggregation operator	Max, Sum, Min, Mean
Activation type	ReLU, PReLU, Swish, Exp
Activation enabled	True, False

```
1  import torch
2  from torch import Tensor
3  from torch_geometric.nn import GCNConv
4  from torch_geometric.datasets import Planetoid
5
6  dataset = Planetoid(root='.', name='Cora')
7
8  class GCN(torch.nn.Module):
9      def __init__(self, in_ch, hidden_ch, out_ch):
10         super().__init__()
11         self.conv1 = GCNConv(in_ch, hidden_ch)
12         self.conv2 = GCNConv(hidden_ch, out_ch)
13     def forward(self, x:Tensor, edge_index: Tensor):
14         x = self.conv1(x, edge_index).relu()
15         x = self.conv2(x, edge_index)
16         return x
17
18 model = GCN(dataset.num_features, 16, dataset.
       num_classes)
```

Listing 1. An user-defined GNN model using PyG [8]

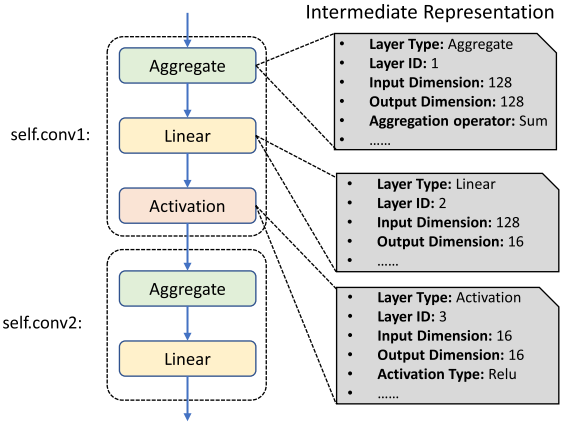

Fig. 7. The computation graph of the GNN in Listing 1.

Aggregate layer: The inputs are the vertex feature vectors $\{\boldsymbol{h}_i^{l-1} \in \mathbb{R}^{f_{\text{in}}} : v_i \in \mathcal{V}\}$ and the edges $\{e : e \in \mathcal{E}\}$. The output feature vector of each vertex is calculated by

$$\boldsymbol{h}_i^l = \text{AggOp}(\boldsymbol{A}_{j,i} \times \boldsymbol{h}_j^{l-1}, j \in \mathcal{N}(i)), \boldsymbol{h}_i^l \in \mathbb{R}^{f_{\text{out}}}, \quad (5)$$

where $f_{\text{in}} = f_{\text{out}}$ and AggOp() is the element-wise Aggregation Operator defined in Table II (e.g., Max, Sum). The Aggregate layer can be executed using SpDMM mode.

Linear Layer: The inputs are the vertex feature vectors $\{\boldsymbol{h}_i^{l-1} \in \mathbb{R}^{f_{\text{in}}} : v_i \in \mathcal{V}\}$ and weight matrix $\boldsymbol{W} \in \mathbb{R}^{f_{\text{in}} \times f_{\text{out}}}$. The output feature vector of each vertex is calculated by

$$\begin{aligned}\boldsymbol{H}_{\text{out}} &= [\boldsymbol{h}_1^l; \boldsymbol{h}_2^l; \ldots; \boldsymbol{h}_{|\mathcal{V}|}^l] = [\boldsymbol{h}_1^{l-1}\boldsymbol{W}; \boldsymbol{h}_2^{l-1}\boldsymbol{W}; \ldots; \boldsymbol{h}_{|\mathcal{V}|}^{l-1}\boldsymbol{W}] \\ &= [\boldsymbol{h}_1^{l-1}; \boldsymbol{h}_2^{l-1}; \ldots; \boldsymbol{h}_{|\mathcal{V}|}^{l-1}]\boldsymbol{W} = \boldsymbol{H}_{\text{in}}\boldsymbol{W}, \end{aligned} \quad (6)$$

where $[\boldsymbol{h}_1^{l-1}; \boldsymbol{h}_2^{l-1}; \ldots; \boldsymbol{h}_{|\mathcal{V}|}^{l-1}]$ is the input feature matrix $\boldsymbol{H}_{\text{in}}$ and $[\boldsymbol{h}_1^l; \boldsymbol{h}_2^l; \ldots; \boldsymbol{h}_{|\mathcal{V}|}^l]$ is the output feature matrix $\boldsymbol{H}_{\text{out}}$. It can be executed using GEMM mode.

Vector-Inner layer: The inputs are the vertex feature vectors $\{\boldsymbol{h}_i^{l-1} \in \mathbb{R}^{f_{\text{in}}} : v_i \in \mathcal{V}\}$ and the edges $e(i,j)$ without edge weight. The output is the weight of each edge calculated by

$$e(i,j).weight = \langle \boldsymbol{h}_i^{l-1}, \boldsymbol{h}_j^{l-1} \rangle, \quad e(i,j) \in \mathcal{E}. \quad (7)$$

Vector-Add Layer: The Vector-Add layer adds feature vectors of two layers. This layer can be used to capture the residue connection design.

Activation Layer: The Activation layer applies the element-wise activation function (e.g., ReLU, PReLU, Swish, Exp) to vertex features or edge weights.

BatchNorm Layer: The input is the feature vector of each vertex $\{\boldsymbol{h}_i^{l-1} \in \mathbb{R}^{f_{\text{in}}} : v_i \in \mathcal{V}\}$. A batch normalization operation [30] is applied to each vertex feature.

B. IR Generation Workflow

The proposed intermediate representation consists of two components: LayerIR and ModelIR. LayerIR is the IR of a computation layer that stores the parameters of a layer, as shown in Table II. ModelIR stores a list of LayerIRs and represents the computation graph corresponding to the target GNN model and the input graph. The implementation of LayerIR and ModelIR is demonstrated in Listing 2.

During compilation, the compiler first translates each computation layer into a LayerIR. Then, all the LayerIRs are connected to form a ModelIR, which represents the computation graph of the input GNN model and the input graph. An example of the IR generation process for the GNN model in Listing 1 is illustrated in Listing 3 (Lines 12–39). Note that for illustration, the example in Listing 3 is an unfolded view of the IR generation process. In the actual implementation, the input parser automatically generates the ModelIR using a `for` loop. After IR generation, the compiler performs compiler optimizations, as shown in Listing 3 (Lines 42–46).

C. Computation Order Optimization

We design the general rule for the computation order optimization. First, we define the *linear operator* in the aggregate layer:

Definition 1. In an Aggregate layer, the aggregation operator AggOp() is a linear operator if AggOp() satisfies the following two properties:

- $\text{AggOp}(\boldsymbol{h}_x + \boldsymbol{h}_y) = \text{AggOp}(\boldsymbol{h}_x) + \text{AggOp}(\boldsymbol{h}_y)$ for any $\boldsymbol{h}_x \in \mathbb{R}^f$ and $\boldsymbol{h}_y \in \mathbb{R}^f$.
- $\text{AggOp}(c\boldsymbol{h}_x) = c\text{AggOp}(\boldsymbol{h}_x)$ for any $\boldsymbol{h}_x \in \mathbb{R}^f$ and any constant c.

```python
from collections import OrderedDict
## The IR of a computation layer
class LayerIR(object):
    def __init__(self):
        self._layertype = None  # Layer Tpe
        self._layerid = 0       # Layer ID
        self._parent_id = []    # Parent Layer IDs
        self._child_id = []     # Child Layer IDs
        self._fin = 0           # Input Dimension
        self._fout = 0          # Output Dimension
        self._nv = 0            # # of vertices
        self._ne = 0            # # of edges
        self._aggoperator=None  # AggOp()
        self._act = None        # Activation type
    def setparameter(self):
        # Setting the parameters for the computation layer
    def complexity(self):
        # Return theoretical computation complexity of the computation layer

## The IR of a GNN model
class ModelIR(object):
    def __init__(self):
        self._layers= OrderedDict()
        self._graphs=None
        self._numl = 0
    def addlayers(self, layer):
        self._layers[layer._layerid] = layer
        self._numl = self._numl + 1
    def orderoptize(self):
        # Step 1: computation order optimization
    def layerfusion(self):
        # Step 2: layer fusion
    def datapartition(self):
        # Step 3: data partitioning
    def kernelMapping(self):
        # Step 4: kernel mapping
    def taskScheduling(self):
        # Step 4: task scheduling
    def codeGeneration(self):
        # Generating Instruction sequence
```

Listing 2. The implementation of LayerIR and ModelIR

For example, Sum() is a linear operator while Max() is a non-linear operator.

Then, we identify the exchangeability of computation order in Theorem 1:

Theorem 1. For a pair of adjacent Aggregate layer and Linear Layer, if the Aggregation operator AggOp() of the Aggregate layer is a *linear operator*, we can exchange the computation order of the Aggregate layer and Linear Layer.

Proof. The computation process of the adjacent Aggregate layer and Linear layer can be expressed as

$$\boldsymbol{h}_i^l = \text{AggOp}(\boldsymbol{A}_{j,i} \times \boldsymbol{h}_j^{l-1} \times \boldsymbol{W}, j \in \mathcal{N}(i)), \tag{8}$$

where AggOp() is the aggregation operator of the Aggregate layer and the $\boldsymbol{W}$ is the weight matrix of the Linear layer. Since the operator AggOp() is a linear operator, the above equation can be written as

$$\boldsymbol{h}_i^l = \text{AggOp}(\boldsymbol{A}_{j,i} \times \boldsymbol{h}_j^{l-1}, j \in \mathcal{N}(i)) \times \boldsymbol{W}. \tag{9}$$

Therefore, the computation order of this pair of Aggregate layer and Linear layer can be exchanged without affecting the final result.

The computation order can affect the total computation complexity. The computation complexity (CC) of an Aggregate layer is

$$\text{CC}_{\text{Aggregate}}(f_{\text{in}}, f_{\text{out}}, |\mathcal{E}|) = 2 \cdot f_{in} \cdot |\mathcal{E}|, (f_{\text{in}} = f_{\text{out}}). \tag{10}$$

The computation complexity (CC) of a Linear layer is

$$\text{CC}_{\text{Linear}}(f_{\text{in}}, f_{\text{out}}, |\mathcal{V}|) = 2 \cdot f_{\text{in}} \cdot f_{\text{out}} \cdot |\mathcal{V}|. \tag{11}$$

```python
dataset = 'Cora'
path = osp.join('.', 'data', dataset)
dataset = Planetoid(path, dataset, transform=T.NormalizeFeatures())
data = dataset[0]

nedges = data.edge_index.shape[1]
nvertices = data.x.shape[0]
nflen = data.x.shape[1]
edge_index = data.edge_index
edge_index = torch.transpose(edge_index, 0, 1)

## IR generation
GNN1 = ModelIR()
GNN1._graphs = data

aggregate1 = LayerIR()
aggregate1.setparameter(
    layertype = 'Aggregate', layerid = 1, parent_id = [], child_id = [2], fin = nflen, fout = nflen, nv = nvertices, ne=nedges, aggoperator=2, act=None, actenable=False, batchenable=False)
GNN1.addlayers(aggregate1)

linear1 = LayerIR()
linear1.setparameter(
    layertype = 'Linear', layerid = 2, parent_id = [1], child_id = [3], fin = nflen, fout = 16, nv = nvertices, ne=nedges, aggoperator=None, act=None, actenable=False, batchenable=False)
GNN1.addlayers(linear1)

activation1 = LayerIR()
activation1.setparameter(
    layertype = 'Activation', layerid = 3, parent_id = [2], child_id = [4], fin = 16, fout = 16, nv = nvertices, ne=nedges, aggoperator=None, act='ReLU', actenable=True, batchenable=False)
GNN1.addlayers(activation1)

aggregate2 = LayerIR()
aggregate2.setparameter(
    layertype = 'Aggregate', layerid = 4, parent_id = [3], child_id = [5], fin = 16, fout = 16, nv = nvertices, ne=nedges, aggoperator=2, act=None, actenable=False, batchenable=False)
GNN1.addlayers(aggregate2)

linear2 = LayerIR()
linear2.setparameter(
    layertype = 'Linear', layerid = 5, parent_id = [4], child_id = [], fin = 16, fout = 7, nv = nvertices, ne=nedges, aggoperator=None, act=None, actenable=False, batchenable=False)
GNN1.addlayers(linear2)

## IR optimizations
GNN1.orderoptize() # Step 1: computation order optimization
GNN1.layerfusion() # Step 2: layer fusion
GNN1.datapartition()  # Step 3: data partitioning
GNN1.kernelMapping()  # Step 4: kernel mapping
GNN1.taskScheduling() # Step 4: task scheduling
GNN1.codeGeneration('GNN1.ga')   # Generating Instruction sequence
```

Listing 3. The example of IR generation

Algorithm 5: Computation Order Optimizaiton.

Input: IR of input GNN model, L: number of layers in IR
Output: Optimized IR
1: **for** $l \leftarrow 1$ to L **do**
2: # Sequentially check the following conditions
3: Check: If layer l has only one child layer: layer m
4: Check: If layer m has only one parent layer: layer l
5: Check: If layer l, m is a {Aggregate, Linear} pair
6: Check: If the operator of the Aggregate layer is linear
7: Check: If exchanging layer l, m reduces computation
8: complexity
9: # Perform conditional computation order exchange
10: **if** all the above conditions are met **then**
11: Exchange layer l and layer m in IR

Suppose the feature vector to the Aggregate-Linear pair (An Aggregate layer followed by a Linear layer) has length f_1, the output feature vector has length f_2. The computation complexity of this Aggregate-Linear pair is

$$\text{CC}_{\text{Aggregate-Linear}} = 2 \cdot f_1 \cdot |\mathcal{E}| + 2 \cdot f_1 \cdot f_2 \cdot |\mathcal{V}|. \quad (12)$$

If the Aggregate layer and the Linear layer is exchangeable, the computation complexity after the exchange is

$$\text{CC}_{\text{Linear-Aggregate}} = 2 \cdot f_1 \cdot f_2 \cdot |\mathcal{V}| + 2 \cdot f_2 \cdot |\mathcal{E}|. \quad (13)$$

Theorem 2. Based on (12) and (13), if $f_1 > f_2$, Linear-Aggregate execution order has lower complexity. If $f_2 > f_1$ Aggregate-Linear execution order has lower complexity. if $f_1 = f_2$, Aggregate-Linear execution order and Linear-Aggregate execution order have the same computation complexity.

Based on Theorems 1 and 2, we propose the computation order optimization as shown in Algorithm 5 (Section VI-C). We iteratively apply Algorithm 5 until no layers can be exchanged.

D. Layer Fusion

After computation order optimization, the compiler performs layer fusion consisting of two types: Activation Fusion and BatchNorm Fusion.

Activation Fusion: An Activation layer can be merged into its adjacent layer, including Aggregate layer, Linear layer, Vector-Inner layer, or Vector-Add layer. Through Activation Fusion, no independent Activation layer is required, which eliminates the external memory traffic between this Activation layer and its adjacent layer.

BatchNorm Fusion: For inference, the coefficients $(\mu, \sigma, \epsilon, \gamma, \beta)$ in the element-wise batch normalization operation are fixed: $y = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}} \cdot \gamma + \beta$. Moreover, the batch normalization operation is a linear operator. Therefore, the BatchNorm layer can be merged with the adjacent Linear layer. The Linear layer incorporates the batch normalization operation into its weights and bias. After BatchNorm Fusion, the BatchNorm layer is eliminated, which reduces total computation complexity and external memory traffic. After layer fusion, the number of computation layers and the computation order of the layers are determined.

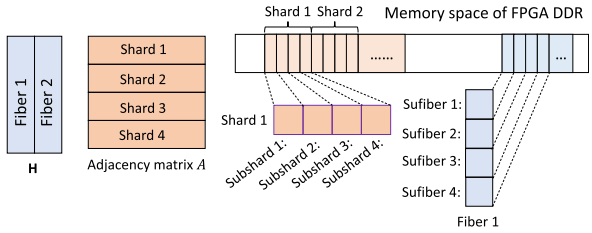

Fig. 8. Data partitioning and memory mapping.

E. Data Partitioning

In real-world applications, input graphs can be very large. The compiler performs data partitioning for each layer, starting from the first layer to the last layer. We propose the Fiber-Shard data partitioning (Fig. 8) to fit the available on-chip memory. In each layer, the graph has an adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ and a feature matrix $\boldsymbol{H} \in \mathbb{R}^{|\mathcal{V}| \times f}$ that need to be partitioned. $\boldsymbol{A}$ contains all the edges and is partitioned to shards along the row dimension. Each shard has N_1 rows and is partitioned into subshards, with each subshard having N_1 columns. The edges in a subshard are stored sequentially in DDR memory, and the subshards in a shard are stored in the contiguous region of DDR memory, as shown in Fig. 8. The Feature matrix $\boldsymbol{H}$ is partitioned into fibers along column dimension, and each fiber is assigned N_2 columns. Each fiber is further partitioned into subfibers, and each subfiber has N_1 rows. For simplicity, $\boldsymbol{A}(i, j)$ denotes the subshard j of shard i. $\boldsymbol{H}(i, j)$ denotes the subfiber j of fiber i. The same partitioning configuration (N_1, N_2) is applied to each layer. The proposed partitioning strategy enables the proposed partition-centric execution scheme (Algorithms 6, 7, and 8), which further ensures that the outputs of a layer maintain the same partitioning configuration (N_1, N_2) as the input. Therefore, the outputs of a layer can be directly used as the input for the next layer since each layer has the same partitioning configuration. Therefore, no data re-partitioning is required between layers.

Partition-Centric Execution Scheme: Based on the Fiber-Shard data partitioning, we propose the partition-centric execution scheme that the execution of a layer is decomposed into a sequence of operations that operate on the data tiles (subshard or subfiber). For example, the execution of an Aggregate layer is described in Algorithm 6. The proposed partition-centric execution scheme leads to reduced memory traffic and random memory access. For the detailed theoretical and empirical analysis of executing the Aggregate layer, please see our previous work [6]. The proposed partition-centric execution scheme has the following benefits: (1) it enables our block-based kernel mapping (Section VI-F) where each Tiling Block can be executed by a PE independently, and there is no data dependency among Tiling Blocks within a layer, and (2) it enables the unified dynamical task scheduling for each computation layer (Section VI-F).

Data Partitioning of a Linear Layer: A Linear layer performs matrix multiplication of input feature matrix $\boldsymbol{H}_{\text{in}} \in \mathbb{R}^{|\mathcal{V}| \times f_{in}}$ and weight matrix $\boldsymbol{W} \in \mathbb{R}^{f_{\text{in}} \times f_{\text{out}}}$. Output feature matrix is $\boldsymbol{H}_{\text{out}} = \boldsymbol{H}_{\text{in}} \boldsymbol{W}$. For the Linear layer, we perform the standard block matrix multiplication. For the Linear layer, the data partitioning keeps the same partitioning configuration as described

Algorithm 6: Partition-Centric Execution Scheme of an Aggregate Layer.

Input: $\boldsymbol{A}$, $\boldsymbol{H}_{\text{in}}$, partitioning configuration (N_1, N_2)
Output: $\boldsymbol{H}_{\text{out}}$

1: *Execution of an Aggregate layer* — Layer Block
2: **for** $i \leftarrow 1$ to $\frac{f_{\text{in}}}{N_2}$ **do**
3: **for** $j \leftarrow 1$ to $\frac{|V|}{N_1}$ **do**
4: **if** there is an idle PE: PE_p **then**
5: Assign $\boldsymbol{H}_{\text{out}}(i,j)$ to PE_p — Tiling Block
6: Initialize $\boldsymbol{H}_{\text{out}}(i,j)$
7: **for** $k \leftarrow 1$ to $\frac{|V|}{N_1}$ **do**
8: load $\boldsymbol{A}(j,k)$ to Edge Buffer
9: load $\boldsymbol{H}_{\text{in}}(k,i)$ to Feature Buffer
10: $\boldsymbol{H}_{\text{out}}(i,j) \leftarrow \text{SpDMM}(\boldsymbol{A}(j,k), \boldsymbol{H}_{\text{in}}(k,i))$
11: Apply activation if required
12: Store $\boldsymbol{H}_{\text{out}}(i,j)$

Algorithm 7: Partition-Centric Execution Scheme of a Vector-Inn Layer.

Input: $\boldsymbol{A}_{\text{in}}$, $\boldsymbol{H}_{\text{in}}$
Output: $\boldsymbol{A}_{\text{out}}$

1: *Execution of a Vector-Inn layer* — Layer Block
2: **for** $i \leftarrow 1$ to $\frac{|V|}{N_1}$ **do**
3: **for** $j \leftarrow 1$ to $\frac{|V|}{N_1}$ **do**
4: **if** there is an idle PE: PE_p **then**
5: Assign $\boldsymbol{A}_{\text{out}}(i,j)$ to PE_p
6: Initialize $\boldsymbol{A}_{\text{out}}(i,j)$ — Tiling Block
7: load $\boldsymbol{A}_{\text{in}}(i,j)$ to Edge Buffer
8: **for** $k \leftarrow 1$ to $\frac{f_{\text{in}}}{N_2}$ **do**
9: load $\boldsymbol{H}_{\text{in}}(i,k)$ to Feature Buffer
10: load $\boldsymbol{H}_{\text{in}}(j,k)$ to Feature Buffer
11: $\boldsymbol{Z} \leftarrow \text{SDDMM}(\boldsymbol{A}_{\text{in}}(i,j), \boldsymbol{H}_{\text{in}}(i,k), \boldsymbol{H}_{\text{in}}(j,k))$
12: $\boldsymbol{A}_{\text{out}}(i,j) \leftarrow \text{Apply}(\boldsymbol{Z})$
13: Apply activation if required
14: Store $\boldsymbol{A}_{\text{out}}(i,j)$

Algorithm 8: Partition-Centric Execution Scheme of an Vector-Add Layer.

Input: $\boldsymbol{H}_{\text{in}}^{l_1}$, $\boldsymbol{H}_{\text{in}}^{l_2}$
Output: $\boldsymbol{H}_{\text{out}}$

1: *Execution of a Vector-Add layer* — Layer Block
2: **for** $i \leftarrow 1$ to $\frac{f_{\text{in}}}{N_2}$ **do**
3: **for** $j \leftarrow 1$ to $\frac{|V|}{N_1}$ **do**
4: **if** there is an idle PE: PE_p **then**
5: Assign $\boldsymbol{H}_{\text{out}}(i,j)$ to PE_p
6: load $\boldsymbol{H}_{\text{in}}^{l_1}(i,j)$ to Feature Buffer — Tiling Block
7: load $\boldsymbol{H}_{\text{in}}^{l_2}(i,j)$ to Feature Buffer
8: $\boldsymbol{H}_{\text{out}}(i,j) \leftarrow \text{VecAdd}(\boldsymbol{H}_{\text{in}}^{l_1}(i,j), \boldsymbol{H}_{\text{in}}^{l_2}(i,j))$
9: Store $\boldsymbol{H}_{\text{out}}(i,j)$

Algorithm 9: Task Scheduling.

Input: $\boldsymbol{A}$ and $\boldsymbol{H}_{\text{in}}$ of input graph; weight matrices; L: number of Layer Blocks.
Output: output embedding of each vertex
for $l \leftarrow 1$ to L **do**
  Load CSI of Layer Block l to Scheduler
  for each Tiling block in Layer Block l **parallel do**
    if there is an idle PE: PE_p **then**
      Assign this Tiling Block to PE_p
      PE_p Executes this Tiling Block
  Wait until all the Tiling Blocks are executed

on Section VI-E for the input feature matrix $\boldsymbol{H}_{\text{in}}$ and output feature matrix $\boldsymbol{H}_{\text{out}}$. The basic computation kernel of a Linear layer is the GEMM.

Data Partitioning of a Vector-Inn Layer: A Vector-Inn layer is to sample the results using adjacent matrix $\boldsymbol{A}_{\text{in}}$ from $(\boldsymbol{H}_{\text{in}}\boldsymbol{H}_{\text{in}}^T)$, which is denoted as $\boldsymbol{A}_{\text{in}} \odot (\boldsymbol{H}_{\text{in}}\boldsymbol{H}_{\text{in}}^T)$. The output $\boldsymbol{A}_{\text{out}}$ is the combination of $\boldsymbol{A}_{\text{in}}$ and the weight value of each non-zero position in $\boldsymbol{A}_{\text{in}}$. The Vector-Inn layer exploits the same partitioning strategy (See Section VI-E) as the Aggregate Layer. The execution scheme of a Vector-Inn layer is shown in Algorithm 7.

Data Partitioning of a Vector-Add Layer: The inputs to a Vector-Add layer are two input feature matrices of the same size – $\boldsymbol{H}_{\text{in}}^{l_1}$ and $\boldsymbol{H}_{\text{in}}^{l_2}$. The output feature matrix $\boldsymbol{H}_{\text{out}}$ is the addition of two matrices: $\boldsymbol{H}_{\text{out}} = \boldsymbol{H}_{\text{in}}^{l_1} + \boldsymbol{H}_{\text{in}}^{l_2}$. The execution of the Vector-Add layer is shown in Algorithm 8.

F. Kernel Mapping and Task Scheduling

Kernel Mapping: Through data partitioning, each layer in the IR is expressed as nested loops (e.g., Algorithm 6) according to the proposed partition-centric execution scheme. The compiler maps each layer to a sequence of high-level instructions. The kernel mapping is performed hierarchically. Each layer is mapped to a block of instructions called *Layer Block* (e.g., Algorithm 6). Each Layer Block contains a Control and Scheduling Instruction (CSI) and a set of *Tiling Blocks*. The Tiling Blocks are generated by unfolding the outer nested loops of a Layer Block. For example, for an Aggregate layer, the generated CSI contains the information of Line 2–3 in Algorithm 6, and $\frac{f_{\text{in}}}{N_2} \times \frac{|V|}{N_1}$ Tiling Blocks are generated by unfolding the outer loops. A Tiling Block has an inseparable sequence of high-level instructions that will be executed by a PE.

Task Scheduling: As shown in Algorithm 9, GraphAGILE executes the GNN inference layer by layer. For each Layer Block, the Scheduler loads the heading Control and Scheduling Instruction (CSI). Then, the Scheduler assigns the Tiling Blocks to the idle PEs, forming a *dynamic load balancing strategy*. Each PE maintains a 1-bit output port to indicate its current status (Idle/Busy). When all the Tiling Blocks within a layer are completely finished, GraphAGILE starts to execute the next layer. Within each Tiling Block, the computation instructions and memory read/write instructions are interleaved. Therefore, we exploit the double buffering technique to overlap the computation and data communication. Specifically, Instruction Decoder & Control Signal Generator needs to issue new memory read instructions when the old computation instruction is not finished, which may incur Write after Read (WAR) data hazard.

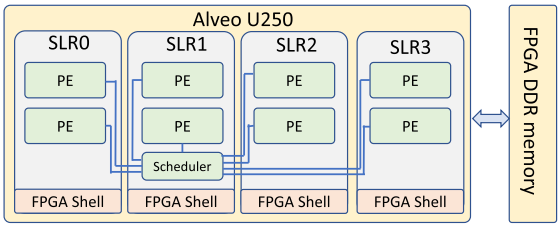

Fig. 9. The mapping of GraphAGILE on Alveo U250.

Therefore, each buffer in a PE maintains a hardware mutex implemented as a one-bit register. After a memory read instruction loads data to a buffer, it locks the mutex of this buffer. After the computation instruction finishes using the data from this buffer, the mutex is unlocked. When a memory read instruction is stalled by a lock, the Instruction Decoder & Control Signal Generator will stop issuing new instructions. Locking/unlocking the mutex is annotated in the high-level instructions by the compiler. Such annotation is through scanning the data dependency among high-level instructions within each Tiling Block, which has negligible complexity. After kernel mapping and mutex annotation, the compiler generates the executable file.

VII. Implementation Details

We conduct comprehensive experiments to evaluate the performance of GraphAGILE. Section VII introduces the implementation details and experimental settings, while Section VIII presents the detailed experimental results, including the execution time (e.g., end-to-end latency, latency of compilation, and latency of hardware execution) (Section VIII-A), the impact of compiler optimizations (Section VIII-B), cross-platform comparison (Section VIII-C), and the comparison with state-of-the-art accelerators (Section VIII-D).

We implement the hardware design on a state-of-the-art FPGA platform, Xilinx Alveo U250, consisting of four Super Logic Regions (SLRs). The FPGA DDR memory has four channels with 77 GB/s memory bandwidth. On U250, we implement 8 PEs where each SLR contains 2 PEs of $p_{sys} = 16$ as shown in Fig. 9. We develop GraphAGILE using Verilog HDL. We synthesize the design and perform Place&Route using Xilinx Vivado 2021.1 to obtain the frequency and resource utilization report. GraphAGILE on Alveo U250 consumes 778 K LUTs (45%), 10,240 DSPs (83%), 1853 BRAMs (69%) and 1,050 URAMs (82%). GraphAGILE runs at 300 MHz. Then, we build a cycle-accurate simulator for the hardware accelerator to evaluate its performance. We use Ramulator [31] to simulate the performance of FPGA DDR memory. We develop the compiler using Python. At runtime, the compiler reads the user-defined GNN models (defined using Pytorch Geometric library (PyG) [32]) and input graphs. Then, the compiler generates the binary file for the hardware accelerator and performs preprocessing for the input graph. After that, the binary file, GNN model weights, and propocessed input graph are sent to the FPGA DDR memory through PCIe. For performance simulation, we set the PCIe bandwidth to be 31.5 GB/s which is the same as the baseline CPU-GPU platform for a fair comparison. The Alveo U250 board has four DDR memories, each connected to an SLR, and each DDR memory has capacity of 16 GB. The DDR memory on the Alveo U250 board is sufficient to store the input graphs used in our experiments (Table IV). For example, for the largest graph used in our experiments, Amazon-Products, has a total size of 7.2 GB, including the vertex features and the edges.

Arithmetic Logic Unit (ALU): The proposed ALU can support multiplication, addition, multiply-add operation, comparison (Mux, Min), ReLU activation, PReLU activation. Each PE also has an Activation Unit and the Activation Unit has 16 Activation Elements (See Fig. 11) that work in parallel. The Activation Element supports exponential function $\texttt{exp(x)}$, sigmoid function $\texttt{1}/(\texttt{1} + \texttt{exp(x)})$, division.

Routing Network: The proposed Index Shuffle Network (ISN) and Data Shuffle Network (DSN) are implemented using Bufferfly Network that is proposed in [29]. The structure of the Bufferfly Network is depicted in Fig. 12. The benefits of using this Bufferfly Network are: (1) the Bufferfly Network is hardware efficient that only consumes a small number of ALUs, (2) there are intermediate buffers in the Switches that can buffer data when there is network congestion. As shown in [29], such intermediate buffers lead to high-throughput data routing.

RAW Unit: In SpDMM mode, read-after-write (RAW) data hazard may occur when accumulators in the Gather Unit read the old feature vertex vector from the Feature/Result Buffer. To resolve the RAW data hazard, we implement a RAW Unit before Gather Unit as shown in Fig. 13. In the RAW Unit, there is a RAW detector to detect the RAW data hazard and a small Reorder Buffer (implemented as a FIFO) to cache the input data when RAW is detected. The data in the Reorder Buffer will be sent to the Gather Unit when there is no RAW data hazard.

System Details of Alveo U250: Fig. 11 depicts the structure of an ALU in the ACK. Each PE has an Index Shuffle Network and a Data Shuffle Network. The Index Shuffle Network has 16 input ports and 16 output ports. Each port has 96 bits since an edge is 96-bit (32-bit source index, 32-bit destination index, 32-bit edge weight). The Data shuffle network also has 16 input ports and 16 output ports. Each port has $(16 + 3) \times 32$ bits where 16×32 bits are used for vector features and 3×32 bits are used for edges. Both Index Shuffle Network and Data Shuffle Network are implemented using a butterfly network. Each PE has an Edge Buffer of size 2 MB, a Feature Buffer of size 3 MB, and a Weight Buffer of size 1 MB. We exploit the double buffering technique for Edge Buffer and Weight Buffer and the triple buffering technique for Feature Buffer. Such triple/double buffering enables the overlapping of computation and data communication. The dimension of a Weight Buffer is $(N_W = 16384) \times (P_{\text{sys}} = 16)$, the dimension of an Edge Buffer is $(N_E = 65\,K) \times 3$, the dimension of a Feature Buffer is $(N_{F1} = 16384) \times (N_{F2} = 16)$.

Resource Utilization: Table III shows the resource utilization of various FPGA accelerators in the experiments.

VIII. Evaluation Results

Fig. 10 shows the IR of various types of widely-used GNN layers evaluated in our experiments.

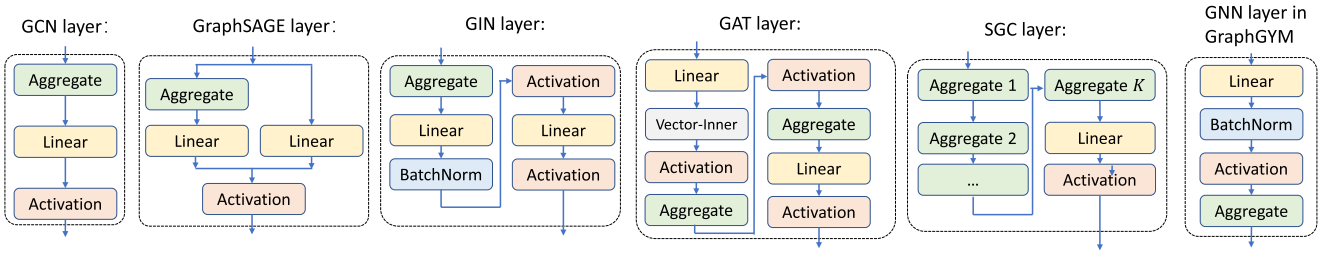

Fig. 10. Intermediate Representations of state-of-the-art GNN layers.

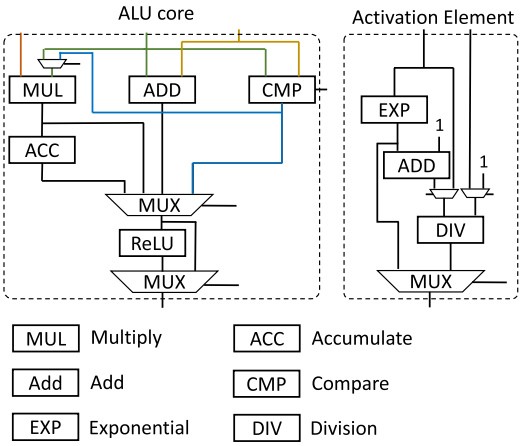

Fig. 11. The structure of the ALU in ACK and the structure of an Activation Element in Activation Unit.

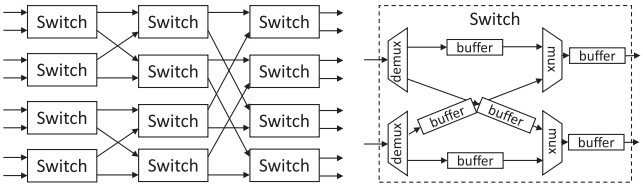

Fig. 12. The structure of ISN and DSN ($p_{sys} = 8$).

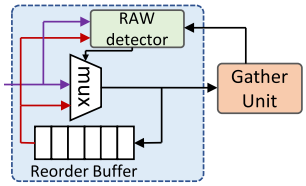

Fig. 13. RAW unit.

TABLE III
SPECIFICATIONS OF FPGA ACCELERATORS IN THE EXPERIMENTS

Platforms	AWB-GCN [11]	BoostGCN [6]	GraphAGILE
Platform	Stratix 10 SX	Stratix 10 GX	Alveo U250
Platform Technology	Intel 14 nm	Intel 14 nm	TSMC 16 nm
Frequency	330 MHz	250 MHz	300 MHz
LUTs/ALMs	200K-300K ALMs	294K ALMs	778K LUTs
DSPs	4096 (Intel DSP)	3840 (Intel DSP)	10240 (Xilinx DSP)
Peak Performance	1351 GFLOPS	640 GFLOPS	614 GFLOPS
On-chip Memory	22MB	32 MB	45 MB
Memory Bandwidth	57.3 GB/s	77 GB/s	77 GB/s

TABLE IV
DATASET STATISTICS

Dataset	Vertices	Edges	Features	Classes
Citeseer (CI) [19]	3327	4732	3703	6
Cora (CO) [19]	2708	5429	1433	7
Pubmed (PU) [19]	19717	44338	500	3
Flickr (FL) [33]	89,250	899,756	500	7
Reddit (RE) [20]	232,965	116,069,919	602	41
Yelp (YE) [33]	716,847	6,977,410	300	100
Amazon-Products (AP) [1]	1,569,960	264,339,468	200	107

TABLE V
EVALUATED GNN MODELS IN THE EXPERIMENTS

Notation	Layer Type	# of layers	Hidden Dimension	Ref.
b1	GCN layer	2	16	[5], [11], [19]
b2	GCN layer	2	128	[5], [11]
b3	GraphSAGE layer	2	128	[20], [33]
b4	GraphSAGE layer	2	256	[20], [33]
b5	GIN layer	5	128-128-128-128	[22]
b6	GAT layer	2	64	[21]
b7	SGC layer	1 (k=2)	N/A	[27]
b8	GraphGym layer	1 preprocessing layer 3 GNN layer 1 postprocessing layer	256	[18]

Baselines: As shown in Table VI, we compare our design with state-of-the-art baselines: CPU-only platform (AMD Ryzen 3990x), CPU-GPU (AMD Ryzen 3990x + Nvidia RTX3090), HyGCN [5], BoostGCN [6], AWB-GCN [11].

Benchmarks: We use eight GNN models in Table V and seven graph datasets[1] in Table IV as benchmarks.

Performance Metric: We evaluate the performance by:

- *End-to-End (E2E) latency T_{E2E}*: The T_{E2E} of GraphAGILE includes (1) the latency of software compilation T_{LoC} on the host processor, (2) the latency of CPU-FPGA data movement T_{comm}, and (3) the latency of executing GNN inference on the accelerator (Latency of hardware execution T_{LoH}). The latency of moving data (processed graph, GNN model, binary file) from host platform to FPGA DDR T_{comm} is estimated through: $T_{\text{comm}} = \frac{\text{total data volume}}{\text{sustained PCIe bandwidth}}$. Then, the end-to-end latency of GraphAGILE is calculated by: $T_{\text{E2E}} = T_{\text{LoC}} + T_{\text{comm}} + T_{\text{LoH}}$.
- *Latency of compilation (LoC) T_{LoC}*: The latency of compilation is the overhead of software or hardware compilation. The measured T_{LoC} of GraphAGILE is the time duration

[1]The dataset Reddit in Table IV is from a pre-existing publicly available third party dataset [20].

TABLE VI
SPECIFICATIONS OF PLATFORMS

Platforms	CPU	GPU	HyGCN [5]	AWB-GCN [11]	BoostGCN [6]	GraphAGILE
Platform	AMD Ryzen 3990x	Nvidia RTX3090	ASIC	Stratix 10 SX	Stratix 10 GX	Alveo U250
Platform Technology	TSMC 7 nm	TSMC 7 nm	TSMC 12 nm	Intel 14 nm	Intel 14 nm	TSMC 16 nm
Frequency	2.90 GHz	1.7 GHz	1 GHz	330 MHz	250 MHz	300 MHz
Peak Performance	3.7 TFLOPS	36 TFLOPS	4608 GFLOPS	1351 GFLOPS	640 GFLOPS	614 GFLOPS
On-chip Memory	256 MB L3 cache	6 MB L2 cache	35.8 MB	22MB	32 MB	45 MB
Memory Bandwidth	107 GB/s	936.2 GB/s	256 GB/s	57.3 GB/s	77 GB/s	77 GB/s

TABLE VII
END-TO-END LATENCY, LATENCY OF COMPILATION, LATENCY OF HARDWARE EXECUTION

Model	Latency (ms)	Dataset						
		CI	CO	PU	FL	RE	YE	AP
b1	T_{E2E}	2.129	0.808	2.126	9.97	128.3	62.9	442.0
	T_{LoC}	0.249	0.215	0.574	2.68	51.1	18.8	263.8
	T_{LoH}	0.320	0.103	0.272	1.28	15.6	11.6	37.4
b2	T_{E2E}	4.364	1.535	4.28	20.1	208.5	155.1	718.1
	T_{LoC}	0.254	0.226	0.66	2.6	49.7	18.3	261.4
	T_{LoH}	2.550	0.819	2.34	11.5	97.2	104.3	315.9
b3	T_{E2E}	4.355	1.574	4.25	21.19	212.7	134.3	657.4
	T_{LoC}	0.235	0.258	0.59	2.58	49.1	19.2	272.2
	T_{LoH}	2.560	0.826	2.38	12.60	102.0	82.6	244.4
b4	T_{E2E}	6.912	2.387	6.919	33.88	315.0	278.2	905.2
	T_{LoC}	0.212	0.237	0.599	2.47	50.1	21.3	270.3
	T_{LoH}	5.140	1.660	5.040	25.40	203.3	224.4	494.1
b5	T_{E2E}	14.99	9.23	15.64	91.73	527.6	901.6	1415.5
	T_{LoC}	0.24	0.23	0.56	2.52	50.9	30.1	300.3
	T_{LoH}	13.10	8.51	13.80	83.20	415.1	839.0	974.4
b6	T_{E2E}	3.139	1.201	3.24	17.69	219.2	123.1	680.9
	T_{LoC}	0.249	0.258	0.58	2.69	50.0	18.7	270.9
	T_{LoH}	1.330	0.453	1.38	8.99	107.6	71.9	269.2
b7	T_{E2E}	2.252	0.826	2.285	11.32	368.8	72.1	601.8
	T_{LoC}	0.223	0.235	0.594	2.63	53.8	17.5	261.4
	T_{LoH}	0.469	0.101	0.411	2.68	253.4	22.1	199.6
b8	T_{E2E}	7.98	3.25	13.79	67.65	537.8	548.2	1749.3
	T_{LoC}	0.23	0.24	0.61	2.74	52.2	28.7	283.5
	T_{LoH}	6.19	2.52	11.90	58.90	424.0	487.0	1325.0

from the time the GNN model (defined using PyG API) and the input graph are provided, *to* the time the input graph is processed and the instruction sequence is generated by the compiler. For the design automation frameworks [6], [12], T_{LoC} includes hardware meta compilation, hardware synthesis, Place&Route, and FPGA reconfiguration.

- *Latency of hardware execution (LoH) T_{LoH}*: Latency of hardware execution is the latency of executing the binary code on the hardware accelerator. Before runtime, the GNN model, processed input graph, and binary file are already stored in the FPGA DDR.

A. Execution Time and Size of Binary File

Execution Time: Table VII shows the measured latency of GraphAGILE. We observe that the software compilation time ranges from 2 ms to 300 ms, which is proportional to the size of the input graph. The reason is that data partitioning is the most time-consuming operation with complexity $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$. The design automation frameworks (e.g., DeepBurning-GL [12]) undergo hours of overhead to perform hardware synthesis and Place&Route. Thus, the proposed software compiler is fast and lightweight.

TABLE VIII
THE SIZE (MB) OF THE GENERATED BINARY FILES [ROW 1–8], AND THE SIZE (MB) OF INPUT GRAPHS [ROW 9]

	Dataset						
	CI	CO	PU	FL	RE	YE	AP
b1	0.136	0.053	0.193	0.194	0.228	0.161	0.246
b2	0.141	0.057	0.234	0.270	0.234	0.218	0.369
b3	0.210	0.084	0.340	0.393	0.340	0.310	0.518
b4	0.217	0.093	0.421	0.421	0.427	0.423	0.764
b5	0.297	0.131	0.632	0.633	0.703	0.661	1.231
b6	0.145	0.060	0.263	0.299	0.264	0.258	0.457
b7	0.204	0.079	0.281	0.281	0.334	0.230	0.342
b8	0.101	0.059	0.422	0.422	0.439	0.528	1.098
Input graph	47	12.6	38	181	1863	900	4223

Size of Binary File: Table VIII shows the size of the generated binary files. Compared with the sizes of input graphs or the inter-layer intermediate results, the size of binary files is negligible. Therefore, loading the binary files from the FPGA external DDR memory to the on-chip scheduler results in a small amount of memory traffic. The size of the binary files is small because the high-level instructions are compact and powerful; For example, a single high-level instruction (128 bits) can define the computation task of a large data partition (up to 16,384 vertices).

B. Impact of the Optimizations

To show the effectiveness of the proposed optimizations, we compare T_{LoH} of using the compiler optimizations and T_{LoH} without compiler optimizations. Figs. 14, 15, and 16 show the impact of (1) computation order optimization, (2) layer fusion, and (3) overlapping the computation and data communication (in task scheduling), respectively.

Computation Order Optimization: Computation order optimization leads to 82%, 9.6%, 9.9%, 6.3%, 1.3%, 121%, 260%, 0% average speedup on **b1-b8**, respectively. The computation order optimization can reduce both the computation complexity and external memory traffic of the involved Aggregate layers. The computation order optimization has no effect on model **b8**, because model **b8** uses a preprocessing MLP layer to transform the feature vectors to a uniform length, which eliminates the opportunities for computation order optimization. Note that the

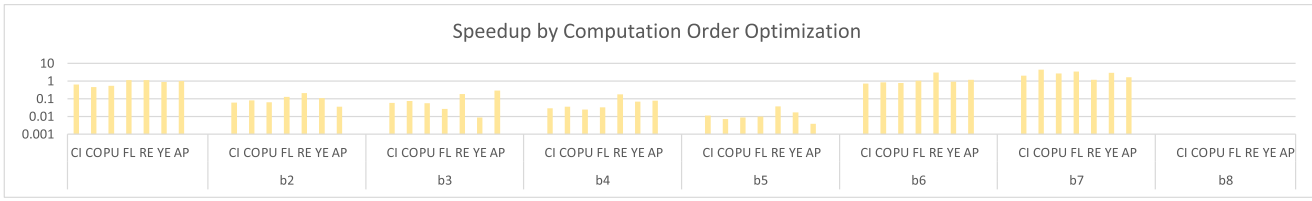

Fig. 14. Impact of computation order optimization on the latency of hardware execution (LoH) T_{LoH}.

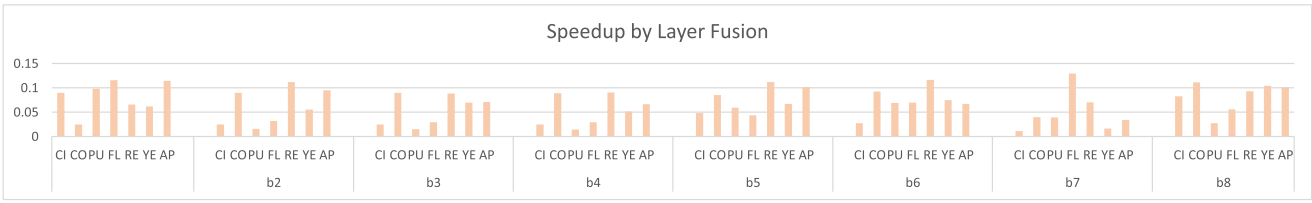

Fig. 15. Impact of layer fusion on the latency of hardware execution (LoH) T_{LoH}.

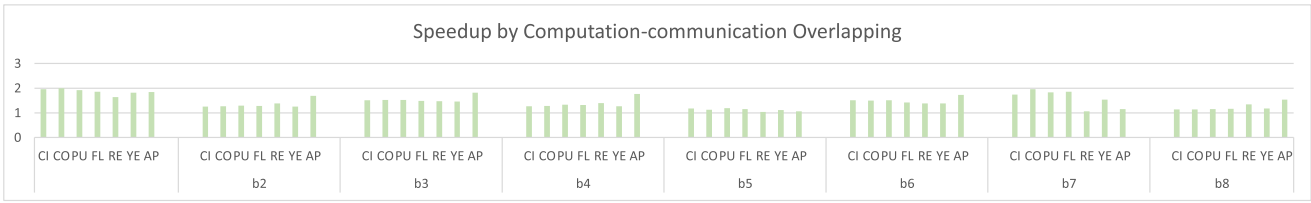

Fig. 16. Impact of computation and communication overlapping on the latency of hardware execution (LoH) T_{LoH}.

Computation order optimization itself has a small overhead ($\approx 0.5\mu s$ average latency) during the software compilation.

Layer Fusion: Layer fusion leads to 8.1%, 6.0%, 5.5%, 5.2%, 7.3%, 7.4%, 4.7%, 8.2% average speedup on **b1-b8**, respectively. The performance improvement is because the individual Activation layers and BatchNorm layers are eliminated (See Section VI-D). Thus, extra memory traffic of the Activation and BatchNorm layers is eliminated to reduce the latency of hardware execution. Note that layer fusion has complexity $\mathcal{O}(L)$ and incurs small overhead ($\approx 0.66\mu s$ average latency) during the software compilation.

Overlapping Computation and Communication: Overlapping the computation and communication leads to 186%, 134%, 153%, 137%, 112%, 148%, 158%, 123% average speedup on **b1-b8**, respectively. It demonstrates the effectiveness of proposed double/triple buffering techniques and the effectiveness of the software compilation optimizations.

C. Cross Platform Comparison

We compare T_{E2E} on three baseline platforms: (1) CPU-only platform, (2) CPU (Ryzen 3990x) + GPU, (3) CPU (Ryzen 3990x) + GraphAGILE. On CPU-only platform, we execute CPU version of Pytorch Geometric (PyG) and Deep Graph Library (DGL), with Intel MKL as the backend. On CPU-GPU platform, we execute GPU version of PyG and DGL, with CUDA 11.3 as the backend. The E2E latency of CPU-only and CPU-GPU platforms include the preprocessing overhead of runtime systems (e.g., GPU kernel launch). Figs. 17 and 18 show the comparison. Compared with PyG-CPU, GraphAGILE achieves $10.3\times - 47.1\times$ speedup on **b1-b8**. Compared with PyG-GPU, GraphAGILE achieves $1.27\times - 3.8\times$ speedup on **b1-b8**. Compared with DGL-CPU, GraphAGILE achieves $9.1\times - 20.1\times$ speedup on **b1-b7**. Compared with DGL-GPU, GraphAGILE achieves $1.7\times - 3.9\times$ speedup on **b1-b7**.

The speedup over CPU-only and CPU-GPU platforms is due to: (1) The kernels in GNN (e.g., SpDMM, SDDMM) have irregular computation&memory access patterns and low data reuse. GraphAGILE hardware architecture optimizes the data path and memory organization for various GNN computation kernels. The processors in CPU or GPU have limited cache sizes (e.g., 32 KB L1 cache and 512 KB L2 cache). The data exchange (due to low data reuse) among L1, L2, and L3 caches becomes the performance bottleneck and results in reduced sustained performance. On CPU platforms, loading data from the L3 cache incurs latency of 32 ns, and loading data from L2 cache incurs latency of $5 - 12$ns. Compared with the CPU-only/CPU-GPU, the ACK in GraphAGILE can access data in one clock cycle from the on-chip edge/weight/feature buffers. Therefore, although the baseline CPU-only and CPU-GPU platforms have higher ($6\times$) peak performance than the state-of-the-art FPGAs, GraphAGILE still outperforms the baselines. (2) The compiler of GraphAGILE automatically performs various optimizations

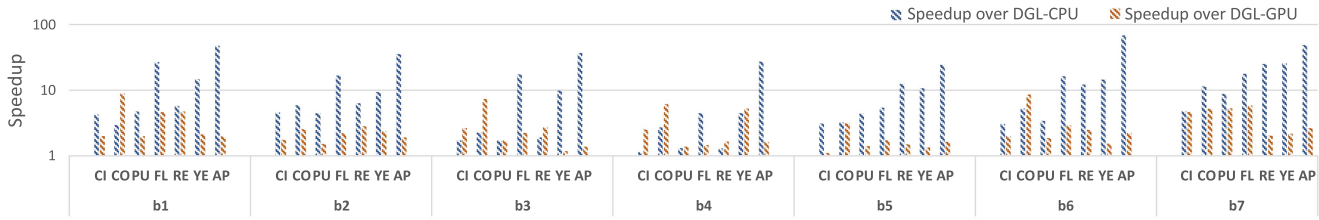

Fig. 17. Comparison of end-to-end latency T_{E2E} with DGL.

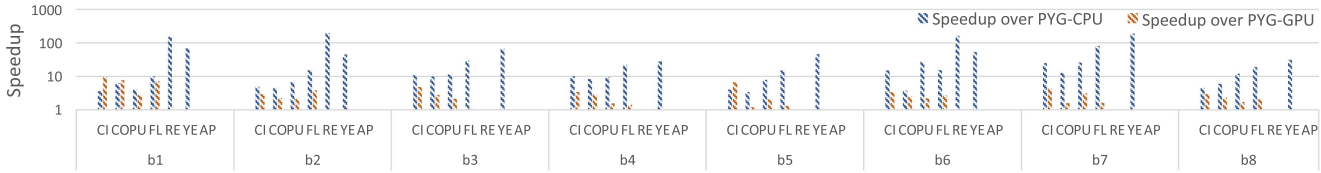

Fig. 18. The comparison of end-to-end latency T_{E2E} with PyG. Note that PyG-CPU cannot execute AP due to out of memory. PyG-GPU cannot execute RE, YE, and AP due to out of memory. Therefore, these results are not shown in the Figure.

TABLE IX
ADVANTAGES OF GRAPHAGILE OVER THE STATE-OF-THE-ART WORK

	GAT	NHC ⋆	Preprocesssing	UFH ‡	GEMM	SDDMM
HyGCN [5]	No ☹	No ☺	graph partitioning, sparsity elimination	No ☹	YES ☺	NO ☹
AWB-GCN [11]	No ☹	No ☺	graph partitioning, data layout transformation	YES ☺	NO ☹	NO ☹
DeepBurning-GL [12]	No ☹	Yes (6-8 hours) ☹	(Unknown)	NO ☹	YES ☺	NO ☹
BoostGCN [6]	No ☹	Yes (6-8 hours) ☹	graph partitioning	NO ☹	YES ☺	NO ☹
GraphAGILE	YES ☺	No ☺	software compilation	YES ☺	YES ☺	YES ☺

⋆ NHC: if the design needs to regenerate hardware if the GNN model or input graph is changed.
‡ UFH: if the design uses the unified hardware module to execute various computation kernels.

to minimize execution time. While the computation order optimization and layer fusion can potentially be applied to CPU-only and CPU-GPU platforms, other compiler optimizations (such as data partitioning for partition-centric execution schemes, task scheduling for dynamic load balancing) are specific to the proposed overlay architecture. For example, data partitioning relies on an effective and customized memory organization. The hardware architecture and the compiler of GraphAGILE perform synergistically to achieve lower latency.

D. Comparison With the State-of-the-Art Accelerators

We compare with state-of-the-art accelerators: HyGCN [5], AWB-GCN [11], DeepBuring-GL [12] and BoostGCN [6].

Advantages of GraphAGILE: Table IX summarizes the performance comparison. HyGCN [5] and AWB-GCN [11] use fixed hardware designs that only support limited GNN models. For example, they cannot execute GAT due to the lack of support for SDDMM. Moreover, they use additional data-dependent optimizations, such as sparsity elimination (HyGCN). These optimizations can reduce the latency of hardware execution T_{LoH} at the cost of increased end-to-end latency due to the expensive preprocessing. Design automation frameworks such as DeepBurning-GL [12] and BoostGCN [6] need to pay hours of overhead to regenerate FPGA bitstream for every pair of GNN models and input graph. Therefore, they have very large end-to-end latency. HyGCN, DeepBurning-GL, and BoostGCN are hybrid architectures that initialize different hardware modules for various computation kernels. However, hybrid architectures suffer from load imbalance and thus, hardware under-utilization. AWB-GCN uses the same set of processing elements to execute SpDMM under various data sparsity. It is not efficient for GEMM and does not support SDDMM. For dense input graphs (e.g., AmazonProducts) or GNN models with the PReLU or SWISH activation functions, GEMM is essential to be supported.

Comparison of Latency of Hardware Execution T_{LoH}: Since no previous work measure the end-to-end latency, their overhead of graph preprocessing (Table IX) are unknown. Therefore, we are only able to compare the latency of hardware execution T_{LoH}, as shown in Table X. Table III shows the detailed resource utilization of various FPGA accelerators. Compared with BoostGCN, GraphAGILE achieves $1.01\times - 2.51\times$ speedup on FL, RE, YE, and AP under comparable peak performance and memory bandwidth. Compared with HyGCN, GraphAGILE achieves $2.97\times$ speedup on RE. GraphAGILE achieves higher performance because BoostGCN and HyGCN are hybrid accelerators that suffer from load imbalance. AWB-GCN is $1.96\times$ faster than GraphAGILE on RE because (1) the platform of AWB-GCN has $2.2\times$ peak performance than GraphAGILE, and (2) AWB-GCN exploits the sparsity of vertex features to reduce the total computation complexity. However, the sparsity

TABLE X
COMPARISON OF T_{LoH}

Model	Dataset	Approach	T_{LoH} (ms)	Speedup
b2	FL	BoostGCN [6]	20.1	1.75×
		GraphAGILE	11.5	1×
	RE	BoostGCN [6]	98.1	1.01×
		HyGCN [5]	289	2.97×
		AWB-GCN [11]	49.7	0.51×
		GraphAGILE	97.2	1×
	YE	BoostGCN [6]	193	1.85×
		GraphAGILE	104.3	1×
	AP	BoostGCN [6]	793.5	2.51×
		GraphAGILE	315.9	1×

exploitation in AWB-GCN requires a runtime system to obtain the sparsity of the intermediate results and dynamically perform data format transformation and kernel remapping. Therefore, the runtime optimizations of AWB-GCN are orthogonal to our static compiler optimizations. For an overlay accelerator, it is challenging to exploit the data sparsity because both data format and high-level instructions need to be generated/changed dynamically at runtime. We leave the dynamic data sparsity optimizations in the runtime system as future work.

IX. Discussion

In real-world applications, the input graphs can be very large, consisting of billions of vertices and edges. For example, the ogbn-papers100M [1] dataset requires more than 100 GB DDR memory to store the full input graph, which is beyond the capacity of the DDR memory on the state-of-the-art FPGA boards (e.g., Xilinx Alveo U250 board has 64 GB on-board DDR memory). GraphAGILE can be easily extended to perform GNN inference on large-scale input graphs. To achieve this, the following are required: (1) coarse-grained data partitioning by the compiler, and (2) a runtime system to perform task scheduling and inter-data-partition communication on the host processor. During compilation, the compiler first partitions the input graph into super data partitions, each fitting in half of the total FPGA on-board DDR memory. Using half of the total FPGA on-board DDR memory, we can overlap the computation and CPU-FPGA data communication via double-buffering. Then, following Section VI, the compiler performs fine-grained data partitioning, kernel mapping, and task scheduling for each super data partition. The compiler will generate a binary file for each super data partition. At runtime, the runtime system on the host processor schedules the execution of super data partitions onto the FPGA accelerator. The runtime system also performs inter-data-partition communication by sending the data from other super data partitions (in the host memory) to the super data partition currently on the FPGA accelerator. The computation on the accelerator and the CPU-FPGA data communication can be overlapped to improve the overall performance. We leave the support for the large-scale input graphs as future work.

X. Conclusion and Future Work

In this work, we proposed a domain-specific overlay accelerator for low-latency GNN inference. The proposed accelerator consists of a novel hardware architecture with an instruction set, and a software compiler with various optimizations for latency reduction. The experimental results showed that compared with the state-of-the-art implementations on CPU-only and CPU-GPU platforms, we achieved $47.1\times$ and $3.9\times$ speedup in end-to-end latency. Compared with state-of-the-art GNN accelerators, we achieved up to $2.9\times$ speedup in terms of hardware execution latency. GraphAGILE has supported widely-used GNN models, including the numerous GNN models in GraphGYM.

Future Work: (1) In the future, we intend to extend GraphAGILE to support various GNN minibatch training algorithms and develop a design automation algorithm to quickly generate the overlay accelerator for a given FPGA platform. (2) We also plan to build a runtime system that performs dynamic sparsity exploitation to reduce the hardware execution latency. The runtime system will perform just-in-time (JIT) compilation, which can dynamically map a computation task (e.g., a layer block in Algorithm 6) to a computation kernel (e.g., GEMM or SpDMM) based on the data sparsity.

Acknowledgment

Equipment and support by AMD Xilinx are greatly appreciated.

References

[1] W. Hu et al., "Open graph benchmark: Datasets for machine learning on graphs," 2020, *arXiv:2005.00687*.
[2] A. Lerer et al., "PyTorch-BigGraph: A large-scale graph embedding system," 2019, *arXiv:1903.12287*.
[3] W. Jiang and J. Luo, "Graph neural network for traffic forecasting: A survey," 2021, *arXiv:2101.11174*.
[4] T. Pfaff et al., "Learning mesh-based simulation with graph networks," 2020, *arXiv:2010.03409*.
[5] M. Yan et al., "HyGCN: A GCN accelerator with hybrid architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2020, pp. 15–29.
[6] B. Zhang, R. Kannan, and V. Prasanna, "BoostGCN: A framework for optimizing GCN inference on FPGA," in *Proc. IEEE 29th Annu. Int. Symp. Field-Programmable Custom Comput. Machines*, 2021, pp. 29–39.
[7] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2020, pp. 255–265.
[8] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric," 2019, *arXiv:1903.02428*.
[9] M. Wang et al., "Deep graph library: A graph-centric, highly-performant package for graph neural networks," 2019, *arXiv:1909.01315*.
[10] B. Zhang, H. Zeng, and V. Prasanna, "Hardware acceleration of large scale GCN inference," in *Proc. IEEE 31st Int. Conf. Appl.-Specific Syst. Architectures Processors*, 2020, pp. 61–68.
[11] T. Geng et al., "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *Proc. IEEE/ACM 53rd Annu. Int. Symp. Microarchitecture*, 2020, pp. 922–936.
[12] S. Liang, C. Liu, Y. Wang, H. Li, and X. Li, "DeepBurning-GL: An automated framework for generating graph neural network accelerators," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des.*, 2020, pp. 1–9.
[13] Y.-C. Lin, B. Zhang, and V. Prasanna, "HP-GNN: Generating high throughput GNN training implementation on CPU-FPGA heterogeneous platform," 2021, *arXiv:2112.11684*.
[14] T. Geng et al., "I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *Proc. IEEE/ACM 54th Annu. Int. Symp. Microarchitecture*, 2021, pp. 1051–1063.

[15] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *Proc. IEEE/ACM 57th Des. Automat. Conf.*, 2020, pp. 1–6.
[16] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "OPU: An FPGA-based overlay processor for convolutional neural networks," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 1, pp. 35–47, Jan. 2020.
[17] M. S. Abdelfattah et al., "DLA: Compiler and FPGA overlay for neural network inference acceleration," in *Proc. IEEE 28th Int. Conf. Field Programmable Log. Appl.*, 2018, pp. 411–4117.
[18] J. You, Z. Ying, and J. Leskovec, "Design space for graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 17009–17021.
[19] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.
[20] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1025–1035.
[21] P. Veličković et al., "Graph attention networks," 2017, *arXiv:1710.10903*.
[22] K. Xu et al., "How powerful are graph neural networks?," 2018, *arXiv:1810.00826*.
[23] DPU, 2020. [Online]. Available: https://www.xilinx.com/products/intellectual-property/dpu.html
[24] NVIDIA NVDLA, 2017. [Online]. Available: http://nvdla.org/
[25] T. Moreau et al., "A hardware–Software blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, Sep./Oct. 2019.
[26] B. Zhang, H. Zeng, and V. Prasanna, "Low-latency mini-batch GNN inference on CPU-FPGA heterogeneous platform," 2022, *arXiv:2206.08536*.
[27] F. Wu et al., "Simplifying graph convolutional networks," in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2019, pp. 6861–6871.
[28] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "HitGraph: High-throughput graph processing framework on FPGA," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 10, pp. 2249–2264, Oct. 2019.
[29] Y.-K. Choi et al., "HBM connect: High-performance HLS interconnect for FPGA HBM," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2021, pp. 116–126.
[30] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2015, pp. 448–456.
[31] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Comput. Architecture Lett.*, vol. 15, no. 1, pp. 45–49, Jan.–Jun. 2016.
[32] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.
[33] H. Zeng et al., "GraphSAINT: Graph sampling based inductive learning method," in *Proc. Int. Conf. Learn. Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=BJe8pkHFwS

Bingyi Zhang received the BS degree in microelectronics from Fudan University, in 2017, and the MS degree in integrated circuit engineering from Fudan University. He is currently working toward the PhD degree in computer engineering at the University of Southern California (USC). His research interests include parallel computing, digital signal processing, digital circuit design. His current work is focused on accelerating graph-based machine learning on FPGA.

Hanqing Zeng received the BEng degree in electronic engineering from the University of Hong Kong, in 2016, and the PhD degree in computer engineering from the University of Southern California, in 2022. He is currently a research scientist with Meta AI. His research interests include large scale graph representation learning, parallel and distributed computing, and algorithm-architecture co-optimization for deep learning applications.

Viktor K. Prasanna (Fellow, IEEE) received the BS degree in electronics engineering from Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from Pennsylvania State University. He is Charles Lee Powell chair in engineering with the Ming Hsieh Department of Electrical and Computer Engineering and professor of computer science with the University of Southern California (USC). His research interests include high performance computing, parallel and distributed systems, reconfigurable computing, and embedded systems. He is the executive director of the USC-Infosys Center for Advanced Software Technologies (CAST) and was an associate director of the USC Chevron Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies (Cisoft). He also serves as the director of the Center for Energy Informatics, USC. He served as the editor-in-chief of the *IEEE Transactions on Computers* during 2003–2006. Currently, he is the editor-in-chief of the *Journal of Parallel and Distributed Computing*. He was the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the steering chair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and is the steering chair of the IEEE International Conference on High Performance Computing (HiPC). He received the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University. He received the W. Wallace McDowell Award from the IEEE Computer Society, in 2015 for his contributions to reconfigurable computing. His work on regular expression matching received one of the most significant papers in FCCM during its first 20 years award, in 2013. He is a fellow of ACM, and the American Association for Advancement of Science (AAAS). He is an elected member of the Academia Europea.