Chinese Journal of Electronics
vol. 33, no. 3, pp. 601–622, May 2024
https://doi.org/10.23919/cje.2023.00.135

REVIEW

A Survey on Graph Neural Network Acceleration: A Hardware Perspective

Shi CHEN[1,2], Jingyu LIU[1,2], and Li SHEN[1,2]

1. *School of Computer, National University of Defense Technology, Changsha 410073, China*
2. *Key Laboratory of Advanced Microprocessor Chips and Systems, Changsha 410073, China*

Corresponding author: Li SHEN, Email: lishen@nudt.edu.cn
Manuscript Received April 18, 2023; Accepted August 24, 2023

Abstract — Graph neural networks (GNNs) have emerged as powerful approaches to learn knowledge about graphs and vertices. The rapid employment of GNNs poses requirements for processing efficiency. Due to incompatibility of general platforms, dedicated hardware devices and platforms are developed to efficiently accelerate training and inference of GNNs. We conduct a survey on hardware acceleration for GNNs. We first include and introduce recent advances of the domain, and then provide a methodology of categorization to classify existing works into three categories. Next, we discuss optimization techniques adopted at different levels. And finally we propose suggestions on future directions to facilitate further works.

Keywords — Graph neural networks, Deep learning acceleration, Domain-specific architecture, Hardware accelerator.

Citation — Shi CHEN, Jingyu LIU, and Li SHEN, "A Survey on Graph Neural Network Acceleration: A Hardware Perspective," *Chinese Journal of Electronics*, vol. 33, no. 3, pp. 601–622, 2024. doi: 10.23919/cje.2023.00.135.

I. Introduction

Graph neural networks (GNNs) are emerging deep learning-based methods operating on graph domain. Recently, with wide adoption of graph-structured data in various industries [1], [2] and significant breakthroughs achieved by variants of GNNs [3], [4], GNN-based applications have been utilized to facilitate tasks and activities in both industries and academia, due to its convincing performance and high interpretability [5]. The rapid development and application of GNNs have led to urgent demands for acceleration, since many of their application scenarios are quite sensitive to latency and throughput [6]–[11]. With limited resources for computation and memory storage, the time overhead of training and inference of GNNs can be easily out of range. Existing researches have applied abundant approaches to improve the efficiency of processing GNNs on general hardware devices [12]–[18]. However, evidence is mounting that the irregularity of graph-structured data and the complexity of GNNs' inherent characteristics in both computation and memory access make general-purpose high-performance platforms ill-suited for accelerating either training or inference of GNNs [19]–[21], since most of them are designed for common applications with regular computation and memory accesses. To be exact, the sparsity of graph-structured data, i.e., non-zero numbers only account for a small portion of adjacency matrix, and power-law distribution, i.e., degrees of vertices in a graph differs significantly, make it almost impossible for GNNs to benefit from memory hierarchy, since there exist few consecutive memory accesses and the spatial and temporal locality are quite weak. Moreover, the power-law distribution can also induce severe workload imbalance, leading to degradation of general-purpose hardware (e.g., CPUs and GPUs), with no regard to their further optimizations [22], [23]. Based on comprehensive analysis of overall execution of graph convolutional networks (GCNs) and stages inside, Yan *et al.* discover potential opportunities to design a dedicated hardware device to accelerate GCNs' inference [20], and propose a hybrid architecture for accelerating inference of GCNs [21], as a real state-of-the-art solution to hardware acceleration for GNNs. From then on, there exist a number of solutions to hardware acceleration for GNNs that adopt various architecture design and optimization techniques [24]–[29].

Existing surveys on the topic of acceleration for

Associate Editor: Prof. Lei XU, Shanghai Jiao Tong University.

GNNs mostly spare more efforts to cover algorithmic approaches and software acceleration on CPUs and GPUs, but pay less attention to hardware acceleration. Liu *et al.* [30] give a review on existing algorithmic acceleration methods for GNNs, including graph-level and model-level optimizations. They have mentioned some hardware techniques and early progress of hardware acceleration for GNNs as future prospects without further illustration and explanation of those researches. By contrast, our article purely target at hardware acceleration for GNNs with an overview of existing hardware approaches and a discussion of their adopted optimization techniques, approaching the topic from a different perspective.

Abadal *et al.* [31] provide a review of the field of GNNs and offer an analysis of software and hardware acceleration schemes, categorizing both software and hardware approaches into three types: software-hardware co-design, graph awareness, and communication-centric design. They investigate several early but representative hardware acceleration approaches, but specific optimization techniques of them are not well explained. To the contrary, our article provide a different methodology only for categorization of hardware approaches based on the architecture of hardware design, including hybrid architecture, holistic architecture, and large-scale architecture. Besides, we track more recent advances of hardware acceleration to illustrate up-to-date trends. Moreover, we separately offer detailed analysis of hardware optimization techniques at different levels, including optimization with relation to memory hierarchy, memory access, computation and processing-in-memory architecture, which is neglected by most existing surveys.

In this article, we provide a review on hardware acceleration for graph neural networks with an overview of existing hardware approaches and an analysis of related hardware optimization techniques. The key contributions of our article can be summarized as follows:

1) We introduce the representative and up-to-date researches on hardware acceleration for GNNs.

2) We propose a methodology for categorization that classifies existing researches into three categories based on their hardware architecture, including hybrid architecture, holistic architecture and large-scale architecture.

3) We introduce the optimization techniques adopted by existing researches at different levels, and conduct concise analyses on them.

4) We discuss characteristics of GNNs and limitations of existing researches, and provide five suggestions on future directions to facilitate further researches.

II. Preliminaries

In this section, we introduce background knowledge about GNNs and offer a brief history of acceleration for GNNs.

1. Graph neural networks

Following previous literature done by Wu *et al.* [32] and Liu *et al.* [33]–[35], we introduce the concept of graph and GNNs. From the perspective of hardware acceleration, the GNNs introduced in this sub-section are exactly representative, since they commonly act as workloads of acceleration for GNNs.

Graph A graph can be typically represented as $G = (V, E)$, in which V is the set of vertices, and E is the set of edges. Given $i, j \in \mathbb{N}$, a vertex can be denoted as $v_i \in V$, and $e_{ij} = (v_i, v_j) \in E$ denotes an edge from v_j to v_i. $N(v) = \{u \in V | (v, u) \in E\}$ denotes the neighborhood of a certain vertex v, containing several vertices connected to v by edges. The adjacency matrix $\boldsymbol{A}$ is a $n \times n$ matrix with $\boldsymbol{A}_{ij} = 1$ if $e_{ij} \in E$ and $\boldsymbol{A}_{ij} = 0$ if $e_{ij} \notin E$. Given $n = |V| \in \mathbb{N}^+$ is the number of vertices in a graph, the matrix of vertex features can be denoted as $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, in which $\boldsymbol{x}_v \in \mathbb{R}^d$ represents the feature vector of a certain vertex and $d \in \mathbb{N}^+$ is the dimension of a vertex feature vector. Given $m = |E| \in \mathbb{N}^+$ is the number of edges in a graph, the matrix of edge features can be denoted as $\boldsymbol{X}^e \in \mathbb{R}^{m \times c}$, in which $\boldsymbol{x}^e_{v,u} \in \mathbb{R}^c$ is the feature vector of an edge (v, u) and c is the dimension of a edge feature vector. GNNs are exactly approaches to acquire knowledge about the structure or topology of a graph and properties of vertices in it by extending the methods of conventional deep learning to the domain of graph.

The k-th layer of general GNNs [3] can be depicted as the following equation (1), which contains two major execution phases. Although various alternatives exist for both $\text{AGGREGATE}^{(k)}(\cdot)$ and $\text{COMBINE}^{(k)}(\cdot)$, their patterns of computation and memory access are distinct [20]. The phase of $\text{AGGREGATE}^{(k)}(\cdot)$ is similar with graph processing and exhibits irregularities, while the phase of $\text{COMBINE}^{(k)}(\cdot)$ is similar with conventional neural networks and exhibits regularities, since the feature vectors are updated with multi-layer perceptrons (MLPs). Some variants of GNNs may look quite different, but all of them exactly adopt such a pattern where $\text{AGGREGATE}^{(k)}(\cdot)$, i.e., aggregation, and $\text{COMBINE}^{(k)}(\cdot)$, i.e., combination, take place in an alternative and iterative manner.

$$\begin{aligned} \boldsymbol{a}_v^{(k)} &= \overset{(k)}{\text{AGGREGATE}}(\{\boldsymbol{h}_u^{k-1} : u \in N(v)\}) \\ \boldsymbol{h}_v^{(k)} &= \overset{(k)}{\text{COMBINE}}(\boldsymbol{h}_v^{k-1}, \boldsymbol{a}_v^{(k)}) \end{aligned} \tag{1}$$

Graph convolutional networks The layer-wise convolution operation of graph convolutional network (GCN) based on spectral methods proposed by Kipf *et al.* [2] can be depicted as (2), where $\boldsymbol{\theta}_0', \boldsymbol{\theta}_1'$ are two free parameters w.r.t. Chebyshev coefficients, and $\boldsymbol{A}, \boldsymbol{D}$ are adjacency matrix and degree matrix of the graph, respectively. $\boldsymbol{g}_{\boldsymbol{\theta}'}$ is a filter parameterized by $\boldsymbol{\theta}' \in \mathbb{R}^N$, i.e., $\boldsymbol{g}_{\boldsymbol{\theta}'} = \text{diag}(\boldsymbol{\theta})$. $\boldsymbol{L}$ is the normalized graph Laplacian $\boldsymbol{L} = \boldsymbol{L}_N - \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}$, and $\boldsymbol{x} \in \mathbb{R}^N$ is an input signal.

$$g_{\theta'} \star x \approx \theta_0' X + \theta_1'(L - I_N)x = \theta_0' x - \theta_1' D^{-\frac{1}{2}} A D^{-\frac{1}{2}} x \tag{2}$$

The definition can be generalized as equation (3), with $\widetilde{A} = A + I_N$, $\widetilde{D_{ii}} = \sum_j \widetilde{A}_{ij}$, $X \in \mathbb{R}^{N \times C}$, $\Theta \in \mathbb{R}^{C \times F}$, and $Z \in \mathbb{R}^{N \times F}$.

$$Z = \widetilde{D}^{-\frac{1}{2}} \widetilde{A} \widetilde{D}^{-\frac{1}{2}} X \Theta \tag{3}$$

The forward model of a two-layer GCN in the literature can be simply depicted as (4). This model is widely adopted as workload of acceleration.

$$Z = f(X, A) = \text{softmax}\left(\hat{A}\,\text{ReLU}(\hat{A} X W^{(0)}) W^{(1)}\right) \tag{4}$$

GraphSAGE [1] is a general inductive framework using spatial-method-based graph convolutional networks. The propagation step of GraphSAGE can be depicted as (5), where W^t is the parameter at layer t, N_v denotes the set of neighbors of vertex v, h_v^t is the embedding of vertex v at layer t.

$$\begin{aligned} h_{N_v}^t &= \underset{t}{\text{AGGREGATE}}\left(\{h_u^{t-1}, \forall u \in N_v\}\right) \\ h_v^t &= \sigma\left(W^t \cdot \left[h_v^{t-1} \| h_{N_v}^t\right]\right) \end{aligned} \tag{5}$$

The AGGREGATE function of GraphSAGE can have various forms. Three aggregator functions as follows are suggested.

- Mean aggregator. The inductive version of the GCN variant can be derived by (6).

$$h_v^t = \sigma\left(W \cdot \text{MEAN}\left(\{h_v^{t-1}\} \cup \{h_u^{t-1}, \forall u \in N_v\}\right)\right) \tag{6}$$

- LSTM aggregator. To operate as aggregators of GraphSAGE, LSTMs are modified to execute on an unordered set by permutating vertex's neighbors.
- Pooling aggregator. A max-pooling operation is applied to the set of the vertex's neighbors, as depicted in (7). Any symmetric function can be an alternative to the max-pooling operation. A number of hardware approaches choose GraphSAGE with pool aggregators, i.e., GraphSAGE-Pool, as workload of acceleration.

$$h_{N_v}^t = \max\left(\{\sigma(W_{\text{pool}} h_u^{t-1} + b), \forall u \in N_v\}\right) \tag{7}$$

Graph recurrent networks Gate mechanism from RNNs like GRU [36] and LSTM [37] are adopted in the propagation step to eliminate the limitation of vanilla GNNs [38] and enhance the effectiveness of the long-term information propagation across the graph.

Li *et al.* [39] propose the gated graph neural networks (GGNNs) with the gate recurrent units (GRUs) in the propagation step. The basic recurrence of the propagation model is depicted as equation (8). A_v is the sub-matrix of the graph adjacency matrix A and denotes the connection of vertex v and its neighbors. Vector a gathers the neighborhood information of vertex v, and z, r are the update and reset gates. Few hardware approaches exactly support acceleration for GGNNs introduced here, but they support GNN workloads that adopts GRUs for temporal-variable graph learning.

$$\begin{aligned} a_v^t &= A_V^{\mathrm{T}}[h_1^{t-1} \; h_2^{t-1} \; \cdots \; h_N^{t-1}]^{\mathrm{T}} + b \\ z_v^t &= \sigma(W^z A_v^t + U^z h_v^{t-1}) \\ r_v^t &= \sigma(W^r a_v^t + U^r h_v^{t-1}) \\ \widetilde{h}_v^t &= \tanh(W a_v^t + U(r_v^t \odot h_v^{t-1})) \\ h_v^t &= (1 - z_v^t) \odot h_v^{t-1} + z_v^t \odot \widetilde{h}_v^t \end{aligned} \tag{8}$$

Graph attention networks Velickovic *et al.* [4] propose the graph attention networks (GATs), following self-attention strategy. The layer-wise computation of the coefficients in the attention mechanism of the vertex pair (i, j) is depicted as equation (9), in which a_{ij} is the attention coefficient of vertex j to i, and N_i denotes the neighborhoods of vertex i in the graph. $h = \{h_1, h_2, \ldots, h_N\}$, $h_i \in \mathbb{R}^F$ represents the input vertex features, and N, F are the number of vertices and the dimension of the features, respectively. Similarly, $h' = \{h_1', h_2', \ldots, h_N'\}$, $h_i' \in \mathbb{R}^{F'}$ represents the output vertex features. $W \in \mathbb{R}^{F' \times F}$ is the weight matrix of a shared linear transformation, $a \in \mathbb{R}^{F'}$ is the weight vector. The final output features of each vertex can be obtained after applying a nonlinearity σ as depicted in (10). Generally, GATs are supported by hardware approaches as variants of GCNs.

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}(a^{\mathrm{T}}[W h_i \| W h_j])\right)}{\sum_{k \in N_i} \exp\left(\text{LeakyReLU}(a^T[W h_i \| W h_k])\right)} \tag{9}$$

$$h_i' = \sigma\left(\sum_{j \in N_i} \alpha_{ij} W h_j\right) \tag{10}$$

Graph isomorphism networks Xu *et al.* [3] propose the graph isomorphism networks (GINs), which can be seen as variants of GCNs. GINs update vertex representations as (11), where $h_v^{(k)}$ is the feature vector of vertex v at the k-th iteration/layer, and $N(v)$ is a set of vertices adjacent to v. ϵ is a learnable parameter or a fixed scalar. Multi-layer perceptrons (MLPs) is utilized to model and learn parameters. Hardware approaches that support GCNs usually support GINs as well.

$$h_v^{(k)} = \overset{(k)}{\text{MLP}}\left((1 + \epsilon^{(k)}) \cdot h_v^{k-1} + \sum_{u \in N(v)} h_u^{k-1}\right) \tag{11}$$

Others Ying *et al.* [40] propose DiffPool, a differentiable graph pooling module that can be adapted to various graph neural network architectures in an hierarchical and end-to-end fashion. They stack L GNN modules and learn to assign nodes to clusters at layer l using em-

beddings generated at layer $l-1$. Given an assignment matrix, the DiffPool module pools nodes at each layer. Given the input adjacency matrix at layer l as $\boldsymbol{A}^{(l)}$, and the input node embedding matrix at layer l as $\boldsymbol{Z}^{(l)}$, suppose the learned assignment matrix $\boldsymbol{S}^{(l)} \in \mathbb{R}^{n_l \times n_{l+1}}$ at layer l has already been computed, the DiffPool layer generate a new adjacency matrix $\boldsymbol{A}^{(l+1)}$ and a new embedding matrix $\boldsymbol{X}^{l+1}$, as depicted in (12) and (13).

$$\boldsymbol{X}^{(l+1)} = \boldsymbol{S}^{(l)^{\mathrm{T}}} \boldsymbol{Z}^{(l)} \in \mathbb{R}^{n_{l+1} \times d} \tag{12}$$

$$\boldsymbol{A}^{(l+1)} = \boldsymbol{S}^{(l)^{\mathrm{T}}} \boldsymbol{A}^{(l)} \boldsymbol{S}^{(l)} \in \mathbb{R}^{n_{l+1} \times n_{l+1}} \tag{13}$$

DiffPool utilize two separate GNNs to generate assignment matrices $\boldsymbol{S}^{(l)}$ and embedding matrices $\boldsymbol{Z}^{(l)}$. The embedding GNN at layer l is a standard GNN module with given inputs, as depicted in (14). The pooling GNN as layer l use the inputs to generate an assignment matrix, as depicted in (15).

$$\boldsymbol{Z}^{(l)} = \underset{l,\text{embed}}{\text{GNN}}(\boldsymbol{A}^{(l)}, \boldsymbol{X}^{(l)}) \tag{14}$$

$$\boldsymbol{S}^{(l)} = \text{softmax}(\underset{l,\text{pool}}{\text{GNN}}(\boldsymbol{A}^{(l)}, \boldsymbol{X}^{(l)})) \tag{15}$$

DiffPool is supported by a number of hardware approaches, though it is not as common as GCNs, GATs, GraphSAGE, and GINs, it can be viewed as representative workloads since it can be easily integrated with existing GNN modules.

Beyond the GNNs introduced above, there exist GNNs that are not commonly supported by hardware approaches [41]–[55]. Those GNNs act as significant workloads for specific approaches, and some of them, such as Cluster-GCN [54] and EdgeConv [46], are SOTA solutions for graph deep learning. Since we focus on hardware approaches of GNN acceleration, for clarity concerns, we don't include the concepts of those GNNs here, but we offer several tables to specify corresponding relationships between hardware approaches and the GNNs.

2. Brief history of graph neural network acceleration

GNNs have achieved state-of-the-art performance in various graph-related tasks [1], [3] and applications, leading to increasing research interests in GNNs. However, as GNNs are widely adopted for solving problems in different domains, it is discovered that the execution efficiency of GNNs is degraded due to real-world factors [30]. The amount of the graph data generated by real-world applications and industries are extraordinarily large, posing non-negligible challenges to both training and inference of GNNs, which is not taken into consideration during algorithmic design initially. Moreover, deeper and more complicated GNNs are utilized as a promising approach to enhance the ability of networks' expression, especially increasing the time overhead of training typical GNNs. Furthermore, a large number of real-world applications [6]–[11], running on either general-purpose or domain-specific processing platforms, pose stringent constraints on latency and throughput. However, with an inappropriate algorithmic design and limited resources for computation and storage, even large-scale and distributed, the time overhead, i.e., latency and throughput related, of GNNs' training and inference can be easily beyond users' expectations. Therefore, with the rapid development of GNNs, accelerating both GNNs' training and inference become an urgent issue.

Existing researches have paid abundant efforts to acquire acceleration methods for a variety of GNNs at algorithm level, not only promoting the model accuracy but also accelerating the model training and inference at the same time [12]–[18]. However, evidence has shown that GNNs' execution on general-purpose platforms can hardly benefit from general optimizations for common applications, due to their unique characteristics [20]. To be exact, GNNs' execution benefits little from the general-purpose memory hierarchy on CPUs and GPUs due to low spatial and temporal locality primarily caused by the sparsity and power-law distribution of the non-zero numbers in adjacency matrix, and different sizes of feature matrices, i.e., irregularity of the data, can lead to degradation of conventional high-performance co-processors, i.e., GPUs and TPUs [56], since most of them are designed for computation of regular-sized data.

Motivated by both the inherent incompatibilities between typical GNN-based applications and the underlying computation platforms, and the urgent demands coming from various industries utilizing GNN-based applications, researchers focus on GNN-specific hardware-based acceleration and achieve distinguished advances. Existing works on hardware-based GNN acceleration mainly use application-specific integrated circuits (ASICs) for implementation of their design, and some of them may use field programmable gate arrays (FPGAs) as a platform for evaluation. Only a few of them really couple FPGAs tightly with their design. With advances of hardware devices, resistive random access memory (ReRAM) is utilized for processing-in-memory (PIM) acceleration. Auten *et al.* [19] propose the first accelerator for GCN inference adopting a straightforward methodology, in which two kinds of processing elements are designed for acceleration of aggregation and combination respectively. Following the similar methodology, HyGCN [21] offers a real hybrid architecture for GCN acceleration, which contains two engines for processing two stages of GCN inference, and applies abundant and hierarchical optimizations to improve the efficiency of single engine and cooperation of the two engines. HyGCN achieves great breakthrough and offers the first state-of-the-art (SOTA) solution for hardware-based GCN acceleration. AWB-GCN [24], EnGN [25], and I-GCN [28] study the execution order of layer-wise matrix multiplication for propagation and offer holistic architecture for GNN acceleration, in which the aggregation and combi-

nation share one group of processing units, typically with preprocessing for data reordering at a considerably low cost. AWB-GCN, equipped with a novel workload rebalancing technique, gives another SOTA solution for GNN acceleration. PIMGCN [27] offers the first PIM-based GCN acceleration using ReRAM crossbars, mainly targeting at solving the incompatibilities between ReRAM crossbars and GCN acceleration, and reducing the crossbar overhead caused by data mapping. PIMGCN achieves relatively high speedup and improvement of energy efficiency compared with existing SOTA solutions, and offers the first SOTA PIM-based design using ReRAM crossbars for GCN acceleration. Beyond the representative works mentioned above, there exist various hardware-based solutions for GNN accelerations adopting distinct optimization techniques and architectural designs. We thoroughly introduce them in following sections, together with their optimization techniques.

III. Categorization

On the basis of the architectural design, existing approaches of hardware acceleration for GNNs can be primarily classified into three categories, including hybrid architectures, holistic architectures and large-scale architectures.

Table 1 provides an overview of the categories and existing hardware approaches of acceleration for GNNs. Generally, all the three categories refer to hardware approaches of acceleration for GNNs that adopt dedicated hardware components in their design. In particular, hybrid architecture and holistic architecture refer to approaches using single chip or device for GNNs working on graph-structured data limited in scale, while large-scale architecture refer to approaches using multiple devices or chips for GNNs working on large-scale and real-world graph data.

Table 1 Categorization and representative hardware acceleration approaches for GNNs

Category	Approaches
Hybrid architectures	[19], [21], [29], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69]
Holistic architectures	[24], [25], [26], [27], [28], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79], [80]
Large-scale architectures	[81], [82], [83], [84], [85]

In the following, we specifically give a definition of each category to facilitate the understanding of the rest of this article, as shown in Definition 1, Definition 2 and Definition 3.

Definition 1 (Hybrid architectures) Hybrid architectures represent hardware approaches which follow the two-phase pattern of modern GNNs depicted in (1) and adopt heterogeneous components on one chip to handle the computation of the two phases mentioned in Section II.1 respectively. The execution of the two components is usually coordinated by a workload-aware pipeline to reduce latency and enhance throughput.

Definition 2 (Holistic architectures) Holistic architectures represent hardware approaches which exploit the common operations of GNNs' layer-wise computation and utilize homogeneous components on one chip to handle the whole computation of specific GNNs mentioned in Section II. This type of approaches usually require extra methods for pre-processing and post-processing to regulate the pattern of computation.

Definition 3 (Large-scale architectures) Large-scale architectures represent hardware approaches which aim at GNNs working on large-scale and real-world graph data, and adopt multiple dedicated chips and devices in a system connected with a topological network to efficiently scatter workloads and reduce the results. This type of approaches usually adopt separate memory management subsystems, or couple special processing units with large-capacity DRAM on CPU side, to accommodate large-scale graph data and improve the efficiency of memory accesses.

1. Hybrid architectures

In this subsection, we introduce representative hardware approaches of GNN acceleration belonging to hybrid architectures defined in Definition 1. Table 2 summarizes existing hybrid architectures, showing GNN applications supported by specific approaches and the implementation technology they are based on.

Auten *et al.* [19] propose an architecture which mainly consists of four modules for different categories of operations during GNN inference, including the graph processing element (GPE) for graph traversals and sequencing computation steps, the DNN accelerator (DNA) for the DNN computation, the aggregator (AGG) for aggregation of the features, and the DNN queue (DNQ) for buffering memory requests and intermediate results. All the modules are connected to a configurable bus so as to support dataflow of different GNN models, and they then together form a tile connected with others by a network on chip (NoC). The GPE is also equipped with a lightweight runtime to coordinate elements within the whole system for workload scheduling and global synchronization. Different sizes of scratchpads and buffers is utilized to store control information and intermediate data.

HyGCN [21] adopts an architecture which includes two engines (see Figure 1), i.e., Aggregation engine and Combination engine, respectively for the computation of aggregation and combination during GCN inference. The Aggregation engine is designed for efficient execution of irregular accesses and computation and is equipped with operation-specific processing units, and the Combination

Table 2 Summary of existing hardware acceleration approaches of hybrid architectures (GS = GraphSAGE)

Approach	Applications	Technology
Auten *et al.* [19]	GCN, GAT, MPNN [41], PGNN [42]	ASIC
HyGCN [21]	GCN, GS, GIN, DiffPool	ASIC
GCoD [29]	GCN, GIN, GAT, GS, ResGCN [43]	FPGA
Zhang *et al.* [57]	GCN	ASIC
`GraphACT` [58]	GCN	CPU-FPGA
DyGNN [59]	GCN, GAT	ASIC
GNNerator [60]	GCN, GS	ASIC
ReGNN [61]	GCN, GS	PIM
GCIM [62]	GCN, GIN, GS	PIM
Chen *et al.* [63]	GCN, GS, G-GCN [44]	ASIC
BoostGCN [64]	GCN	FPGA
DeepBurning-GL [65]	GCN, GS, RGCN [45], EdgeConv [46]	FPGA
Zhu *et al.* [66]	GCN	PIM
H-GCN [67]	GCN	ACAP
HP-GNN [68]	GCN, GS	CPU-FPGA
NTGAT [69]	GAT	ASIC

engine aims to maximize the efficiency of regular accesses and computation, following the design of TPU [56], i.e., an optimized systolic array. A memory access handler is utilized to manage all the requests to the DRAM, and a Coordinator, i.e., a communication interface, is utilized to mitigate the interference and enable pipelining between the two engines. HyGCN adopts abundant buffers to hide the DRAM access latency, exploit the spatial locality and enhance the data reuse. Besides, both intra- and inter-engine optimizations are conducted to further enhance the data reuse, exploit edge-level and vertex-level parallelism, reduce the latency and energy consumption, and coordinate the off-chip memory access of the two engines during execution.

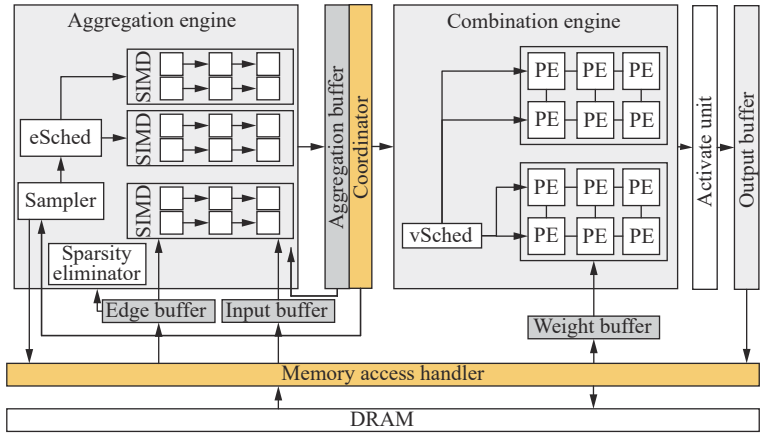

Figure 1 Architecture overview of HyGCN [21].

The hardware architecture proposed by Zhang *et al.* [57] mainly consists of customized hardware modules for aggregation and combination. The execution of the modules is pipelined, and a scheduling strategy is introduced to improve the efficiency and versatility. A flexible datapath is provided to support distinct computation order of GCN inference. Moreover, a data partition scheme and a scheduling strategy based on it is proposed to fully utilize the on-chip storage capacity and improve the efficiency of GCN inference. Following a methodology of algorithm-hardware co-optimization, for reducing off-chip memory accesses, a two-phase graph pre-processing algorithm is utilized, including methods of graph sparsification and vertex re-ordering.

`GraphACT` [58] offers a heterogeneous solution to acceleration of GCN training, in which workload is partitioned between CPU and FPGA, and the computation-intensive parts are offloaded to FPGA while the communication-intensive parts are left for CPU. A processing pipeline containing two main computational modules, i.e., a feature aggregation module and a weight transformation module, is implemented on FPGA. Moreover,

`GraphACT` adopts scheduling strategies between CPU and FPGA, and that of modules on FPGA to exploit the parallelism and improve the throughput of the system.

DyGNN [59] supports dynamic pruning for both vertices and edges, and exploit that mechanism for performance improvement. DyGNN is mainly composed of three types of engines: Aggregator, Updater and Pruner. The Pruner prunes vertices and edges based on a proposed pruning algorithm. The Aggregator conducts aggregation, and the Updater is built with two multi-granular systolic arrays to perform matrix-vector multiplication either separately for edges and vertices or jointly for vertices. A duplication-free mechanism is introduced into the Pruner to fully exploit the data reuse so as to improve efficiency of the execution. A configurable pipelined architecture is also developed to support variants of GNNs, with different execution flow controlled by predefined configurations.

GNNerator [60] provides a programmable design with heterogeneous compute engines, consisting of a Dense engine for dense, regular computations and a Graph engine for sparse, irregular computations. GNNerator manages to exploit the inter-stage parallelism by decoupling the combination and aggregation stages and enabling a flexible execution order of the two engines. The Dense and Graph engines are provisioned to exploit the abundant inter- and intra-vertex parallelism in GNNs. A feature dimension-blocking dataflow with hardware support is introduced to reduce the overhead during feature aggregation stages by exploiting the independence of feature dimensions.

ReGNN [61] uses heterogeneous ReRAM-based PIM units for accelerating GNN inference. The architecture is equipped with aggregation engine for accelerating non-MVM operations, and combination engine for accelerating MVM operations. Aggregation engine, designed with both analog PIM (APIM) and digital PIM (DPIM), consists of sub-engines for aggregation using different operators, including max, sum, and mean, based on the degree of vertices and the dimension of features. A vertex scheduler is also introduced to assign tasks to sub-engines based on the computation parallelism. Moreover, the sub-engines for aggregation adopts novel data mapping strategies to exploit inter- and intra-vertex parallelism. Combination engine, mainly designed with APIM, follows a conventional neural network accelerator with analog PIM crossbars to support intensive and regular MVM operations.

GCIM [62] adopts a 3D-stacked computation-in-memory (CIM) architecture which stacks three types of die, including base die, logic-in-memory (LIM) die, and DRAM die. LIM die is exactly the DRAM die integrated with logic for the aggregation using some lightweight logic units (LLU). The base die is integrated with the combination logic mainly using a systolic array and auxiliary hardware units. Memory-bounded aggregation operations are offloaded to LIM dies near memory banks while the computation-bounded combination operations are offloaded to base dies, which fully exploits bank-level bandwidth and parallelism, and sufficient computational ability. Moreover, a graph partitioning and mapping strategy is introduced to eliminate overhead of data movement and balance the workload, so as to further exploit the data locality and utilize the high bandwidth of the CIM architecture.

GCoD [29] provides a GCN algorithm and accelerator co-design framework, which consists of a split and conquer training strategy to polarize graphs to be either denser or sparser, and a dedicated two-pronged accelerator to leverage algorithm's resulting graph adjacency matrices for improvement of accelerating efficiency. The hardware accelerator is composed of two branches, i.e., two parts, one of which adopts a chunk-based micro-architecture to accelerate the polarized denser sub-graphs with regular and denser patterns and balanced workloads, the other accelerates irregular and sparser but largely reduced sparser workloads.

The design proposed by Chen *et al.* [63] includes a dynamic redundancy-eliminated neighborhood message passing algorithm for GNNs based on a redundancy-aware graph representation, targeting at redundancy of EdgeUpdate and Aggregation. The hardware architecture is designed for the proposed algorithm and it can transform the redundancy elimination into performance improvement. Furthermore, the architecture is also configurable and pipelined so as to support different GNN variants.

BoostGCN [64] proposes a system architecture consisting of external memory and FPGA. The external memory is utilized to accommodate adjacency matrix, weight matrices, and feature matrices. The FPGA board is programmed into heterogeneous processing units, including feature aggregation modules (FAMs) to perform feature aggregation and feature update modules (FUMs) to perform feature update. A internal buffer is utilized to cache the intermediate results produced by FAMs, and memory controller is responsible for handling the data transmissions between external memory and hardware modules.

DeepBurning-GL [65] provides an automatic GNN acceleration framework targeting at specific GNN application by exploiting the reconfiguration capability of FPGAs. The framework relies on a GNN accelerator template and accelerator component templates to initiate the design and generate the accelerator based on optimized design parameters. It contains two categories of GNN computation templates for regular computing, such as the feature extraction and update, and irregular computing, such as graph-based aggregation, respectively. The template for regular computing is exactly a systolic array or a dot-production array while the template for irregular computing is an array of homogeneous processing units. The two kinds of templates can be customized to meet the requirements of GNNs' different phases to maximize the computing efficiency. The memory template is

utilized to generate the design of on-chip memory blocks to buffer data for computation, and a graph manipulation template is adopted to sample from a large graph, or construct graph from raw data, such as point cloud.

Zhu *et al.* [66] propose a parallelism enhancement framework for PIM-based GCN architectures, composed of a algorithmic GCN quantization method to transform the 32-bit floating-poiont graph data to the fixed-point form to reduce hardware overhead, and a RRAM-based multi-core PIM architecture for GCNs called RP-GCN, with an aggregation core array and a combination array to exploit cluster-level computation parallelism. The two categories of arrays form a coarse-grained pipeline dataflow to improve the throughput.

H-GCN [67] proposes a GCN acceleration architecture based on Xilinx Versal ACAP architecture [86], which is an emerging heterogeneous compute platform with strong heterogeneity. To fully exploit the capability of ACAP, H-GCN mix sparse/dense systolic tensor arrays to accelerate the hybrid pattern of GCNs. The architecture generally consists of a sparse-dense matrix-matrix multiplications (SpMM) unit and a PL controller in programmable logic (PL) and a sparse/dense systolic tensor arrays in AI engines (AIEs). The PL controller controls SpMM unit to cooperate with the sparse/dense systolic tensor array to perform all GCN computation. Specifically, with a strategy of input graph reordering, the feature aggregation of the vertices in the dense rectangular areas and in the remaining areas are mapped onto AIEs and the SpMM unit in the PL respectively.

HP-GNN [68] provides a framework for generation of high throughput GNN training implementations on a given CPU-FPGA platform. The framework takes GNN training algorithms and GNN models as inputs, and performs hardware mapping onto the target CPU-FPGA platform. Sampling is executed on CPU since CPU is flexible to support various sampling algorithms. Other operations including feature aggregation and feature update are performed on the proposed FPGA accelerator with the support of Aggregate kernels and Update kernels implemented on the board.

NTGAT [69] offers an acceleration architecture dedicated to the acceleration for GATs together with a runtime node tailoring algorithm and a pipelining insertion sorting scheme. The algorithmic strategies reduce the workloads for computation afterwards. The architecture can be divided into a GAT convolution kernel and a dense computing kernel. The dense computing kernel processes the full connection layer and another linear transformation in which vector inner-production is performed. The GAT convolution kernel is composed of graph encoder, attention coefficients chunks, node engines, and feature cache, and it's responsible for graph-based attention-like computation.

2. Holistic architectures

In this sub-section, we introduce hardware approaches belonging to holistic architectures as defined in Definition 2. Table 3 summarizes holistic architectures and provide corresponding relations between specific approaches, the GNN applications supported by the approaches, and the implementation technology they are based on.

Table 3 Summary of existing hardware acceleration approaches of holistic architectures (GS = GraphSAGE)

Approach	Applications	Technology
AWB-GCN [24]	GCN	ASIC
EnGN [25]	GCN, GS, Gated-GCN [47], GRN [48], R-GCN [45]	ASIC
GCNAX [26]	GCN	FPGA
PIMGCN [27]	GCN, GS, GIN	PIM
I-GCN [28]	GCN, GS, GIN	ASIC
Cambricon-G [70]	GCN, GS, DiffPool, DGMG [49], EdgeConv [46], GUN [50]	ASIC
SGCNAX [71]	GCN	FPGA
GNNIE [72]	GCN, GAT, GS, GIN, DiffPool	ASIC
ReFlip [73]	GCN, GAT, GS, GIN	PIM
ReaDy [74]	CD-GCN [87], TGCN [51], MPNN-LSTM [52]	PIM
Rubik [75]	GS, GIN	ASIC
PASGCN [76]	GCN, GS, GIN	PIM
FusedGCN [77]	GCN	FPGA
G-CoS [78]	GCN, GAT, LGCN [53], GS	FPGA
LW-GCN [79]	GCN, GS	FPGA
DARe [80]	Cluster-GCN [54]	PIM

AWB-GCN [24] treats the layer-wise forward propagation of a multi-layer spectral GCN as a two-step sparse matrix multiplication (SpMM) and adopts an alternative computation order to exploit sparse-dense matrix multiplications and reduce the number of operations during computation. Therefore, AWB-GCN adopts a

baseline architecture to accelerate SpMM kernels in a column-wise-product manner with optimizations including inter- and intra-layer pipelining and data forwarding, matrix blocking and data mapping. Based on the proposed architecture, AWB-GCN is further equipped with strategies at three levels of granularity, i.e., distribution smoothing, row switching, and row remapping, to cope with the workload imbalance.

EnGN [25] is developed based on a unified processing models covering general GNNs. Accompanied with ring-edge-reduce update dataflow, the hardware architecture (Figure 2) includes an array of homogeneous processing units, the same column of which is interconnected with neighbors in a ring network, called ring-edge-reduce array, for operations of aggregations. The on-chip memory hierarchy is coupled with processing element (PE) register file and multi-level caches to alleviate the overhead of memory access. Moreover, dimension-aware stage re-ordering, and graph tiling and scheduling is utilized for optimization based on the characteristics of GNN algorithms. The dimension-aware stage re-ordering changes the computing order of GNNs based on the input and output property dimension comparison. The graph tiling splits vertices of the whole graph into several disjointed partitions so that each row of PE handles features of vertices within the on-chip buffers, and the graph scheduling is used to handle dependencies between tiles so as to fully exploit data reuse to reduce external memory accesses.

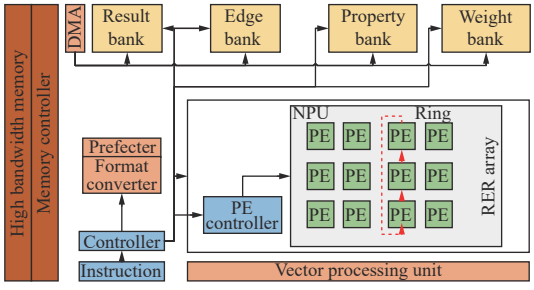

Figure 2 EnGN hardware architecture [25].

The overall architecture (Figure 3) of PIMGCN [27] is extended from a design pattern with both CAM crossbars and MAC crossbars, including a central controller and two engines (search engine and compute engine). The search engine consists of CAM crossbars, following the design of GaaS-X [88]. The compute engine consists of components respectively for aggregation and combination, which are both composed of MAC crossbars. The central controller loads graph data and offloads the GCN results to the external memory, and it controls CAM crossbars, MAC crossbars and special functions units (SFU) that handles the partial results from MAC crossbars. The two engines form a Ping-Pong architecture and run alternatively to process GCN layers per iteration. To maximize the exploitation of inter-vertex parallelism, the execution of destination vertices are parallelized by setting three constraints on the crossbars which ensure that the features of source vertices of different destination vertices locate in different crossbars, the destination vertices to be aggregated locate in different crossbars as well. Moreover, a latency-matching pipeline which writes all the destination vertices after two cycles of aggregation and combination at once to minimize the divergence between read/write latency, and an extra constraint is utilized to ensure two groups of destination vertices locate in different crossbars. To better support inter-vertex parallelism, a GroupCOO format is utilized to store edges and weights.

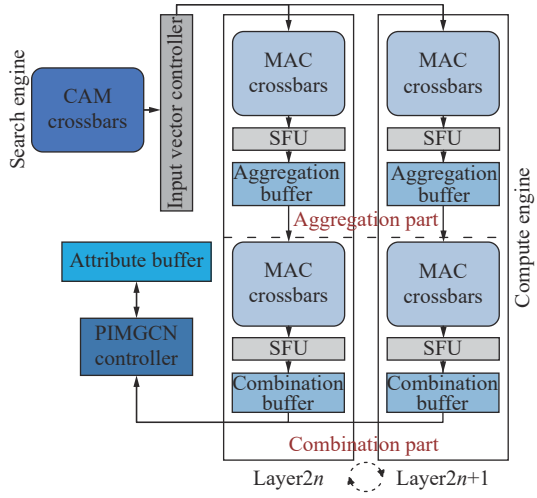

Figure 3 PIMGCN architecture overview [27].

The irregularity and the sparsity of the adjacency matrix can lead to poor data reuse during subsequent computation, which consequently degrades the overall performance. Thus, I-GCN [28] aims to handle the problems induced by the adjacency matrix as much as possible to gain improvement of performance. It adopts an algorithm-hardware co-design methodology and provides a hardware-assisted graph reconstructing algorithm, termed Islandization, to significantly improve data locality and reduce redundant computation. The hardware architecture mainly consists of a Island locator and a Island consumer. The Island locator, primarily supported by a breadth-first search algorithm, works round by round to detect and fetch Islands, i.e., vertices with strong internal connections and week external connections, and usually share neighbors, and Hubs, i.e., vertices with considerably high degree, to feed the Island consumer downstream so as to complete the computation of aggregation and combination. The Island consumer treats the Islands and related Hubs as a small, dense sub-graph and adopts an alternative computation order to complete the aggregation and combination using the same MAC array. Moreover, with such two structures, the Island consumer is able to conduct redundancy elimination with ease and further accelerate the overall computation.

Cambricon-G [70] abstracts the computation of different GNN variants to the process of adjacent cuboid.

The adjacent cuboid combines the vertex feature dimension with the adjacent matrix and multidimensional multilevel tiling is introduced to improve the data reuse and parallelism. Each cuboid can be tiled in three dimensions, including vertex destination, source vertex, and vertex feature, and the size of the partition called cubelet is adapted for that of the on-chip memory. Multidimensional spatial tiling is performed within a cubelet to exploit the data reuse and parallelism, supported by the cuboid engine and hybrid on-chip memory of the proposed architecture. To support multidimensional temporal tiling across cubelets for large GNNs, a programming model is introduced to compute multiple cubelets by specifying the tiling parameters and implementing the computation logic.

GCNAX [26] comes up with an optimized dataflow that is designed to flexibly adapt the loop order and loop fusion techniques for different GCN configurations, with the support of a hardware accelerator. GCNAX adopts an outer-product based method for sparse matrix multiplications in GCNs to alleviate the workload imbalance. Moreover, the compute engine, buffer size and structure are adapted to fit with the execution order and tile sizes of the dataflow. SGCNAX [71] adopts GCNAX as its single PE, aiming to tackle with both inter-PE and intra-PE workload imbalances to better cope with a variety of graphs and GCNs. Since GCNAX has almost adopted optimizations for alleviating the intra-PE workload imbalances, SGCNAX employs a group-and-shuffle approach to conquer the inter-PE workload imbalances.

GNNIE [72] partitions both feature vectors and weight matrix into blocks and schedules the weighting computation in the CPE(computation PE) array. A flexible MAC architecture with adaptive number of MACs per CPE, and dynamic workload redistribution between paired rows of CPEs are adopted by GNNIE to ensure the load-balancing during GNN computation. GNNIE optimizes GAT computation of aggregation, characterizes the dataflow and maps the edge-based computation to the CPE array with the support of SPUs (special functional units) for LeakyReLU and exponentiation, etc. A graph-specific frequency-based caching policy is introduced into GNNIE, ensuring all random accesses are confined to on-chip buffers and all the off-chip accesses are sequential.

ReFlip [73] adopts a unified hardware design to accelerate both the aggregation and combination executions of GCNs by using crossbar-based processing-in-memory architectures. The design of ReFlip mainly consists of a number of processing engines (PEs) connected with a bus [89], each of which is composed of multiple crossbar-based computation units (CUs). ReFlip can execute the combination and aggregation kernels alternatively to complete the computation of all the GCN layers, and it also provides a flexible scheduling orders between the two kinds of kernels in each layer. A layer-wise weight mapping for combination phase and a flipped mapping for aggregation phase is introduced to ReFlip to exploit both inter- and intra-vertex parallelism, improve the utilization of crossbar cells, and reduce the overhead of data accesses. Software-hardware cooperative optimizations, including execution model and storage format, and locality-aware design, are conducted to maximize the efficiency and minimize energy consumption.

ReaDy [74] follows the design of ReFlip [73] and modified it to support the acceleration of dynamic graph convolutional networks (DGCN). Since matrix-vector multiplication (MVM) is the dominant operation in both GCN and RNN kernels within the execution of DGCN, ReaDy is able to utilize homogeneous PEs with crossbar-based computation units to handle all the computations. Based on a vertex-centric mapping strategy, i.e., a flipped mapping strategy of ReFlip, ReaDy further provides redundancy-free scheduling and locality-aware dataflow respectively for GCN and RNN kernels so as to reduce data loads and exploit inter-vertex data reuse. An inter-kernel pipeline with decoupled execution of GCN and RNN kernels is introduced to enhance the computational parallelism and the data reuse of aggregated results of GCN kernels.

Rubik [75] consists of an input scheduling methodology, a mapping methodology, and a hardware architecture design cooperating with the two methodologies. The input scheduling methodology adopts a lightweight graph reordering methods to exploit graph-level locality, and the mapping methodology partitions the input graph and maps inter-vertex and intra-vertex computation to PEs and multiply-and-accumulator (MAC) arrays inside them. The hardware architecture design is tailored from a neural network accelerator to utilize graph-level locality. The on-chip memory hierarchy is well-designed to enhance inter- and intra-PE data reuse and reduce the overhead of memory accesses.

PASGCN [76] adopts the design of PIMGCN [27] as a baseline architecture, together with its two scheduling strategies to exploit inter-vertex parallelism and improve the system throughput. Meanwhile, a lightweight GCN, named AsparGCN, is introduced to remove redundant edges so as to speed up the inference processed by {PIMGCN} as much as possible with insignificant loss of accuracy. AsparGCN includes a number of trainable predictors which learns the edge selection strategy in a layer-wise manner. Based on the prediction results of the predictors, the graph of each layer is sparsified and edges interfering the inter-vertex parallelism are dropped. Despite a little change on the CAM crossbars of {PIMGCN}, {PASGCN} directly uses the resulting sparsified graphs to accelerate GCN inference with almost no hardware overhead.

FusedGCN [77] provides a new systolic architecture that computes the product of the three matrices of GCNs' computation in a combined/fused manner. The architecture can well support the sparsity of graph adjacency matrices and that of input features of the first layer. The

structure of the systolic array and the corresponding dataflow can be unrolled to adapt to the input and output bandwidth.

G-CoS [78] is a GNN and accelerator co-search framework automatically searching for the matched GNN structures and accelerators. The framework integrates a generic GNN accelerator search space applicable to various GNN structures and a on-hot GNN and accelerator co-search algorithm capable of simultaneously and efficiently searching for optimal GNN structures and the matched accelerators. Different from a previous framework DeepBurning-GL [65] which utilizes a heterogeneous template, G-CoS adopts a homogeneous multi-accelerator micro-architecture template to accelerate both the combination and aggregation phases.

LW-GCN [79] proposes a lightweight software and hardware co-optimized accelerator to efficiently perform GCN inference. The sparse matrix is compressed into a packet-level column-only coordinate-list (PCOO) format that can be easily decompressed by hardware. A unified micro-architecture is adopted to execute both combination and aggregation, which are exactly matrix-matrix multiplication (MM) and sparse matrix-matrix multiplication (SpMM). Each PE contains an optimized computation pipeline to alleviate the irregularity in computation and memory accesses caused by SpMM. An additional pre-processing algorithm is utilized to prevent data collisions caused by sparse matrix.

DARe [80] is a manycore architecture for accelerating GNN training by leveraging the benefits of ReRAM-based PEs and efficient on-chip communication supported by a Drop-aware 3D NoC. The Drop-aware 3D NoC, inspired by DropLayer techniques including DropEdge [90] and Dropout [91], is utilized to reduce the communication latency by allocating adequate number of ReRAM PEs to each layer. Each ReRAM-based PE contains multiple crossbars for MAC operations and a router for data exchange. The DropLayer is implemented in the NoC-based architecture by a reconfigurable linear feedback shift register (LFSR) based control mechanism to decide which data to drop per epoch.

3. Large-scale architectures

This sub-section introduces hardware approaches belonging to large-scale architectures as defined in Definition 3. Table 4 summarizes large-scale architectures and offer an overview of GNN applications supported by them, and the implementation technology they are based on. Note that both the applications and the implementation technology are a little bit different from the other two categories.

Graphite [81] provides a combination of a number of cooperative software-hardware techniques to tackle memory problems during execution of GNNs on CPUs, which are mainly motivated by the potential benefits of running GNNs on CPUs, related to the demands of datacenters. Software optimizations, including parallel vector-

Table 4 Summary of existing hardware acceleration approaches of large-scale architectures (GS = GraphSAGE)

Approach	Applications	Technology
Graphite [81]	GCN, GS, etc.	CPU-DMA
SmartSAGE [82]	GS, etc.	CPU-ISP
Li *et al.* [83]	GS + DSSM [55], etc.	FaaS
GNNear [84]	GCN, GIN, GS, GAT	NMP
MultiGCN [85]	GCN, GIN, GS	MultiAccSys

ized aggregation, layer fusion, feature compression, and temporal locality improvement are adopted to relieve the DRAM bandwidth pressure during both training and inference of GNNs on multi-core CPUs. To further improve the performance and reduce stalls during aggregation caused by memory accesses, DMA engines (Figure 4) are modified to support a DMA-aggregation algorithm. The hardware aided aggregation can work cooperatively with most of the software optimizations mentioned above, and the overall execution is pipelined.

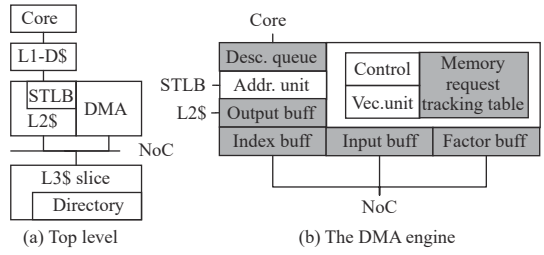

Figure 4 Graphite's enhanced DMA engine [81].

SmartSAGE [82] is an in-storage-processing (ISP) GNN training system targeting at problems induced by the scaling up of both graph datasets and GNNs. SmartSAGE is able to intelligently offload the data intensive stages to ISP units coupled closely inside the SSD. The architecture adopts a ISP accelerator modified to support sub-graph generation, cooperative with a latency-optimized runtime system and host driver.

Li *et al.* [83] aim to provide a practical and promising solution to handle the problems of large scale distributed GNN (LSD-GNN) at hyperscale. They propose a customized scalable and programmable hardware architecture to solve LSD-GNN's problems, and integrate the hardware with an industrial framework. Moreover, they utilize FPGA-as-a-Service (FaaS) together with their customized hardware as a solution to achieve accessibility, scalability, and flexibility. Furthermore, they provide suggestions for future FaaS system designs based on extensive exploration of a variety of FaaS system architecture and their proposed solution.

GNNear [84] harnesses both near-memory processing (NMP) and centralized processing to achieve high-throughput, energy-efficient and scalable GNN training on large-scale graphs. Specifically, DIMM-based near-memory engines (NMEs) and a centralized acceleration engine (CAE) are both adopted to process the memory-

intensive reduce operations and computation-intensive update operations respectively. Optimization strategies are proposed to tackle with resource under-utilization and load-imbalance problems to improve training throughput, concerning data reuse, data mapping, graph partition, and dataflow scheduling.

MultiGCN [85] is an efficient MultiAccSys (multi-node acceleration system) that accelerates the inference phase of large-scale GCNs by trading network latency for network bandwidth. The architecture is scaled from a single-node accelerator to form a system like tensor processing unit (TPU) Pod [92]. The single-node accelerator serves as a single processing node, and multiple processing nodes of that kind are connected to a topology network to construct the whole system. A topology-aware multicast mechanism with a one put per multicast message-passing model is utilized to alleviate network bandwidth requirements, and a *scatter-based round execution mechanism* works with the multicast mechanism in a cooperative fashion. The graph is partitioned in to subgraphs to reduce redundant off-chip memory accesses.

IV. Optimization Techniques

Researchers have applied optimization techniques to their base architectures mainly from three aspects, including memory hierarchy, memory access, and computation to enhance the processing efficiency. For processing-in-memory architectures, dedicated optimization techniques is adopted to improve the throughput and energy efficiency and lower the hardware overhead.

1. Memory hierarchy optimizations

Memory hierarchy optimizations include modifications applied to the on-chip/off-chip memory system to alleviate architecture-aware bottlenecks and further improve the overall performance, such as dedicated on-chip buffers, graph-aware caches, and so on.

Auten *et al.* [19] utilize buffers for intermediate results and data transmission, different sizes of scratchpads are utilized for control information and (large) data storage. HyGCN [21] employs embedded DRAM to cache various data and introduces different buffers to the two engines (Figure 1). For aggregation engine, edge buffers are utilized to cache edges so as to exploit spatial locality, while input buffers are used to cache the vertex features. Aggregation buffers are used to cache intermediate results of aggregation phase to exploit temporal locality. For Combination engine, weight buffer is utilized to cache weight matrix to exploit temporal locality, and output buffer is used to merge write accesses of final features. Edge buffer, input buffer, weight buffer and output buffer all leverage the double buffer techniques to hide the latency. In a similar way, a number of existing hardware approaches [26], [29], [59], [63]–[66], [68], [71], [77]–[79] employ dedicated buffers with double buffering techniques for input, output and intermediate results in their architecture. AWB-GCN [24] also adopts a scratchpad so as to cache part of adjacency matrix on-chip as much as possible, to reduce off-chip bandwidth requirements.

Different from approaches using various dedicated buffers mentioned, EnGN [25] employs a dedicated three-level on-chip memory hierarchy (Figure 5), including register files, multi-level caches, and result banks, to alleviate the overhead of memory accesses. The register files are equipped with processing elements (PEs), while the multi-level caches, exactly degree aware vertex caches (DAVC) for high-degree vertices only, are inserted between register files and result banks. The result banks are utilized to store temporary aggregation results, as the last level of the on-chip memory hierarchy. Rubik [75] offers another on-chip memory hierarchy, composed of global buffer for PE array, private G-D and G-C cache in each PE, and register files in each MAC.

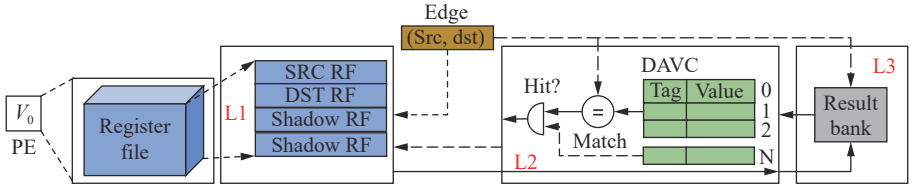

Figure 5 Three-level on-chip memory hierarchy of EnGN [25].

Cambricon-G [70] adopts a software-hardware hybrid memory hierarchy to leverage shared data, which contains the software-controlled scratchpad memory (SPM) for regular data accesses and the hardware-managed topology-aware cache for irregular data accesses. The hardware-managed cache is designed to handle the reuse of source vertices in computational operations, and it is topology-aware to improve the data locality of vertex accesses (Figure 6).

GNNIE [72] adopts a frequency-based caching policy, i.e., graph-specific caching, to maximize the reuse of cached data and reduce off-chip random memory accesses. Contrary to existing frequency-based caching for graphs in software frameworks, e.g., Cagra [93], the caching pol-

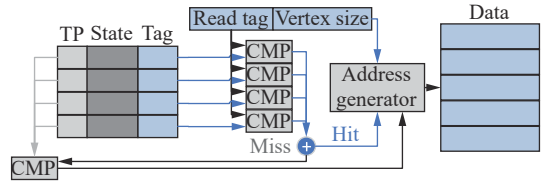

Figure 6 Topology-aware cache of Cambricon-G [70]

icy is hardware-centric dynamic approach with considerably low hardware overhead. Under the proposed caching policy, a set of vertices and edges between them are cached as a sub-graph, and then a partial aggregation is performed in it. Vertices with the most unprocessed edges are most likely to be preserved in the caches after replacement, as depicted in Figure 7, where α, γ denote degree of two vertices. Coupled with an inexpensive preprocessing that bins vertices in order of their degrees so that they are stored contiguously in DRAM in descending degree order of the bins, the graph-specific caching policy is able to avoid random off-chip DRAM accesses and ensure all random accesses are confined to on-chip buffers.

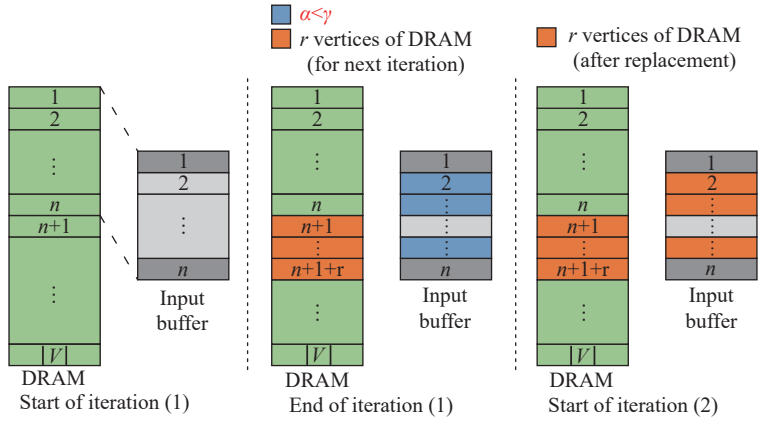

Figure 7 Input buffer replacement policy during aggregation [72]

NTGAT [69] is coupled with a feature oriented set-associate cache. Each cache line of a set stores features of a vertex, and the length and quantity of cache line can be dynamically configured to adapt to various vertex feature size with the same cache capacity, achieving high flexibility using limited resources. The least recently used (LRU) caching policy is introduced to the feature cache.

2. Memory access optimizations

Memory access optimizations involve techniques applied to the architectures so as to regularize off-chip memory access pattern and reduce volume of both off/on-chip memory accesses and data transmission.

EnGN [25] adopts graph tiling to tile large graphs into intervals and shards using a graph partition approach proposed by GridGraph [94]. Given $Q \in \mathbb{N}^+$, vertices are divided into Q disjointed intervals, and edges of the graph with both source and destination vertices limited to one interval can be partitioned into Q^2 shards. EnGN processes with the granularity of a tile, and the size of shards is fitted with the on-chip memory to ensure efficient computation without off-chip memory accesses. An adaptive scheduling approach is adopted to resolve the data dependency among tiles and maintain the sequence of execution considering the structure of a graph.

GCoD [29] adopts a split and conquer algorithm which leverages sub-graph classification to enforce regularity at both fine- and coarse-grained granularity, and utilizes graph partitioning to further boost efficiency. Vertices with similar degrees are clustered into the same class to reduce the irregularity of GCNs' graph adjacency matrices. Each class is then divided into sub-graphs to further achieve a finer-grained regularity. Using graph partitioning, sub-graphs within the same class are uniformly distributed into different groups, reducing the boundary connections to enforce the sparser patterns so as to reduce the irregularity of memory accesses. Similar with GCoD, H-GCN [67] utilizes input graph reordering at training stage to group vertices with more shared neighbors together to improve the data reuse.

GCNAX [26] and SGCNAX [71] adopts local memory promotion and fused matrix multiplication to minimize external data transfer during execution of their proposed chain sparse-dense matrix multiplication. The loop order of consecutive matrix multiplication is changed to reduce redundant memory accesses, and data transfer of intermediate data is eliminated by fusing consecutive execution of matrix multiplications. Off-chip data access is modeled as an optimization problem using several design choices as variables, and thus the overhead caused by off-chip data accesses is minimized by appropriately adopting corresponding design. Equation (16) illustrates the optimization problem used in SGCNAX to achieve optimal computation latency, on-chip SRAM accesses, and off-chip DRAM accesses, where $\mathbb{X} = \mathcal{X}^t \cup \mathcal{X}^{oo} \cup \mathcal{X}^f \cup \mathcal{X}^u \cup \mathcal{X}^{oi}$ denotes the entire search space, and $\mathcal{X}^t, \mathcal{X}^{oo}, \mathcal{X}^f, \mathcal{X}^u, \mathcal{X}^{oi}$ denote the parameter spaces of tile size, inter-tiling loop order, loop fusion strategy, respectively. L, V_d, V_s denote the computation latency, the number of off-chip DRAM accesses and the on-chip SRAM accesses, respectively. $S_X, S_W, S_{B1}, S_A, S_O, S_{B2}$ denote the required on-chip storage size of the corresponding matrices, which are determined by the tile size. ω_1, ω_2 are adjustment parameters that reflect the difference in the energy cost between basic arithmetic operation, DRAM access and SRAM access.

I-GCN [28] utilizes a runtime locality enhancement technique termed Islandization. Each vertex in an island,

labeled as an island vertex, only connects to the those of the same island and the hub vertices connected to the island. This ensures that the space between the L-shapes, i.e. the hubs, is purely blank. Therefore, when processing a GCN layer, the associated data of island vertices are only needed when the island is being processed. Consequently, they only need to be fetched from off-chip once. The hubs do have the chance of being used multiple times during the processing of different islands and inter-hub connections. However, since hubs are normally a small fraction of the entire graph, their associated data will likely be stored on-chip and sufficiently reused. Even if the hubs' associated data is too large to fit in the on-chip memory, Islandization still reduces off-chip data movement. In summary, for GCN processing of real-world graphs with component structures using I-GCN, most data are fetched only once, except the adjacency data of some island vertices which may need to be accessed multiple times during the multi-round island locating.

Rubik [75] employs a lightweight graph reordering methodology that improves the graph-level data locality. The algorithm is developed by using synergistic locality-sensitive hashing (LSH) and row-column ordering. Based on the reordered graph, the intermediate aggregation computation results can be better reused. An alternative heuristic is adopted to efficiently obtain shared vertex sets within adjacent vertices in the execution order to maximize the potential computation results reuse.

Moreover, ReaDy [74] adopts a dedicated scheduling strategy to eliminate redundant data accesses during computation. Zhang *et al.* [57], Cambricon-G [70] both adopt data partitioning to improve the locality of data accesses. GNNerator adopts dimension-blocked dataflows supported by a hardware-aided algorithm to eliminate irregular off-chip memory accesses by exploiting extra on-chip memory accesses, as expressed in Algorithm 1. HP-GNN [68] uses a novel data layout and internal representation to reduce the memory traffic and the number of random memory accesses at training phase.

$$\begin{aligned}\underset{\mathbb{X}}{\text{Minimize}}\ & J = L(\mathcal{X}^u) + \omega_1 \cdot V_d(\mathcal{X}^t, \mathcal{X}^{oo}, \mathcal{X}^f) \\ & \qquad + \omega_2 \cdot V_s(\mathcal{X}^u, \mathcal{X}^{oi}) \\ \text{s.t.}\quad & 0 < T_m \le M, \quad 0 < T_k \le K, \\ & 0 < T_{n0} \le N, \quad 0 < T_{n1} \le N, \\ & 0 < T_{c0} \le C, \quad 0 < T_{c1} \le C, \\ & S_X + S_W + S_{B1} \le \text{GLB}_{\text{size}}, \\ & S_A + S_O + S_{B2} \le \text{GLB}_{\text{size}}, \\ & P_{n0} \times P_{c0} \times P_k \le \#\text{PEs}, \\ & P_{n1} \times P_{c1} \times P_k \le \#\text{PEs}. \end{aligned} \tag{16}$$

Algorithm 1 Dimension-blocking algorithm

Input: sharded graph G; width/height of square shard grid S, hidden dimension size D, features h, layers L.

1: **for** $l \leftarrow 0$ to L **do**
2: **for** blockD $\leftarrow 0$ to D/B **do**
3: **for** dst $\leftarrow 0$ to S **do**
4: **for** src $\leftarrow 0$ to S **do**
5: Shard $\leftarrow G$.Shards(src, dst);
6: **for** $v \leftarrow 0$ to Shard(src, dst).V **do**
7: **for** $u \leftarrow 0$ to $v.U$ **do**
8: **for** $d \leftarrow 0$ to B **do**
9: dim $\leftarrow f(d,$ blockD)
10: $h_{\text{agg}}[v][\text{dim}] \leftarrow \text{Aggregate}(h_u[\text{dim}], h_v[\text{dim}])$;
11: $h'[\text{dst}][:] \leftarrow \text{FeatureExtract}(h_{\text{agg}}[\text{dst}][\text{blockD} \times B : (\text{blockD}+1) \times B], h'[\text{dst}][:])$;
12: $h \leftarrow h'$.

BoostGCN [64] uses data tiling by a low-complexity index-based partion scheme together with a dedicated data storage scheme in external memory while performing partition-centric feature aggregation to optimize on-chip dataflow and off-chip data storage. Vertices are divided into disjoint subsets (internals), and edges are divided into subsets (blocks). Vertex features are divided into slices by 3-D partitioning. The increased edge block size leads to better data reuse within blocks, less computation steps, and lower overall memory access latency with the support of the dedicated data storage scheme, achieving massive parallelism.

The sub-accelerators designed by G-CoS [78] are integrated with functions of weight buffer sharing and buffer re-purposing to further increase on-chip reuse opportunities and reduce off-chip accesses. All the on-chip weight buffers are inter-connected, and the feature, weight and output buffers are inter-changeable, so the off-chip memory accesses are minimized.

LW-GCN [79] adopts a data compression strategy including a novel PCOO format and quantization. The input matrices of the first layer of GCN is compressed to the PCOO format so that only valuable information will be processed afterwards, and storage requirement and computation complexity can be reduced. To further reduce memory consumption, LW-GCN apply quantization onto the values of all the matrices in GCNs. Post-training quantization strategy is utilized to save time for pre-processing since LW-GCN targets at inference of GNNs. The quantization scheme consists of several formats for different matrices. 16-bit signed fixed point (SINT16) is selected to quantize the features and weights. For sparse matrices, 4-bit signed fixed point (SINT4) is selected to quantize the non-zero elements. All the intermediate results are stored as 32-bit signed fixed point (SINT32) to maintain accuracy. Moreover, data collision resolution is introduced to enable several PEs to access a single row in the same memory slices at the same time. A multi-bank memory system is developed to reduce such collision as a muti-port memory to store weights with row grouping and data replication in

the micro-architecture of {LW-GCN}, and empty elements are inserted to further alleviate unresolved collision.

3. Computation optimizations

Computation optimizations involve techniques applied to the hardware architectures to increase overall throughput, and reduce execution latency, including pipelining, eliminating redundant computation and workload balancing.

1) Pipelining

HyGCN [21] forms a execution pipeline of aggregation engine and combination engine, which supports latency-aware and energy-aware functions to reduce the processing latency for each vertex and the energy consumption caused by redundant accesses, as depicted in Figure 8. The data reuse of aggregation results is enhanced and the parallelism of the two engines are improved by decoupling the executions of them, coupled with the ping-pong buffering mechanism.

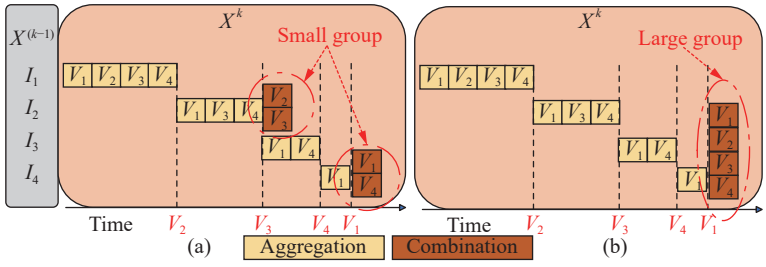

Figure 8 Timing illustration of (a) Latency-aware pipeline and (b) Energy-aware pipeline [21].

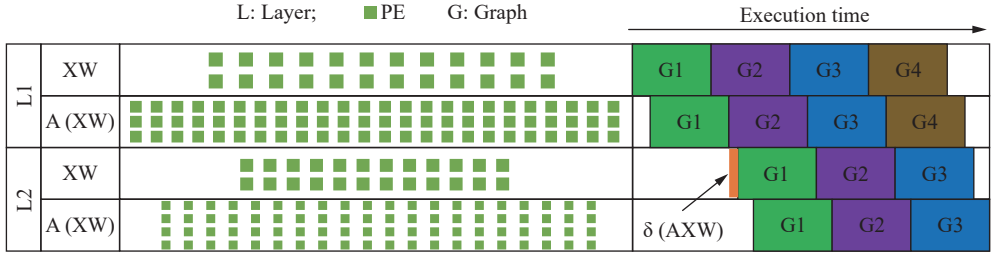

Figure 9 Pipelined SpMMs [24].

AWB-GCN [24] utilizes the pipelining SpMM chains (Figure 9), including intra-layer SpMM pipelining and inter-layer SpMM pipelining to fully exploit the parallelism between consecutive SpMMs to improve the throughput and reduce the latency.

H-GCN [67] exploit the parallelism between consecutive SpMMs in a layer through fine-grained pipelining, with generated STPEs/TPEs, PL, and customized tile size. GNNerator [60] provides fine-grained pipelining of the feature extraction and aggregation stages, in which each of its graph engine or dense engine can be either the producer or the consumer, to further support more variants of GCNs. `GraphACT` [58] utilizes two kinds of pipelines, one of which is between CPU and FPGA, and the other is between modules of FPGA.

ReaDy [74] proposes a inter-kernel pipeline to reuse the aggregated results of GCN kernels. Both DyGNN [59] and Chen *et al.* [63] adopt a flexible pipelined execution flow to support variants of GNNs and reduce the overhead of memory accesses.

BoostGCN [64] utilizes a task scheduling optimization to reduce the pipeline stalls. The internals are sorted by their vertex degrees, and a buffer in external memory is allocated to temporarily store intermediate results to ensure consecutive modules are ready to continue processing. The computation order of the three-matrix multiplication in GCN layer is also considered to achieve high computation efficiency in a pipeline fashion.

NTGAT [69] designs pipelines with two stages in the node engines. The sorting processor and the vector processor share the ping-pong buffer and work simultaneously so as to exploit abundant parallelism.

2) Eliminating redundant computations

HyGCN [21] proposes a dynamic data-aware sparsity elimination to to reduce the redundant accesses since the graph connections are sparsely distributed, utilizing a window-based sliding and shrinking approach. Similarly, I-GCN [28] first pre-aggregates the results of the combination, and then scan the local adjacency bitmap and performs aggregation, after all the combination results of all vertices in an island are ready. If the number of non-zeros in the sliding window exceeds half of the window's size, the features of corresponding vertices will be merged. Otherwise, their connections will be removed from the local adjacency bitmap.

DyGNN [59] adopts an extra module called Pruner to eliminate edge redundancy and vertex redundancy, while Chen *et al.* [63] integrate the redundancy-elimination with

modules (e.g., aggregator and coordinator) to construct several redundancy-eliminated units to prune redundant vertices and edges.

PASGCN [76] uses the lightweight GCN network architecture, named ASparGCN, to remove redundant edges so as to accelerate the inference of GCN inference on PIMGCN [27] accelerator. ASparGCN integrates a parameterized predictor for learning the edge selection strategy for each GCN layer in inference stage. The predictor is able to predict edges preserved for each layer based on the graph structure and input vertex features. Based on the prediction result, the irrelevant edges will be dropped. In the test phase, the resulting sparsified graphs can be used directly in each layer's inference separately, thereby accelerating the total GCN inference. ReaDy [74] provides redundancy-free scheduling for GCN kernels, so that redundant operations can be eliminated.

3) Workload balancing

AWB-GCN [24] utilizes approaches at three levels of granularity to rebalance workloads between processing elements (PEs) round by round. Distribution smoothing balances the workload among neighbors. Remote switching shuffles workloads of regions with the most and least clustered non-zero elements, making efficient distribution smoothing possible. If a row is observed to still contain too many elements to be smoothed or balanced by remote switching, it is designated as an evil row, and then it's partitioned and non-zero elements are remapped to multiple regions (with least clustered elements).

GNNIE [72] utilizes a adaptable MAC architecture and adopts load redistribution. Given $g \in \mathbb{N}^+$, the CPE array is divided into g groups, each of which has equal number of rows. The number of MACs in each CPE, $|\text{MAC}|_i$, is monotonically nondecreasing along the rows: $|\text{MAC}|_1 \leq |\text{MAC}|_2 \leq \cdots \leq |\text{MAC}|_g$. CPE rows are paired and a portion of workload from heavily loaded CPE rows can be offloaded to lightly loaded ones.

SGCNAX [71] utilizes a group-and-shuffle approach to balance workloads among PEs in which the rows of a sparse matrix are grouped by the density-sorted rank order and mapped to PEs so that all the PEs will simultaneously complete the task. An extra *unshuffle* operation executed afterwards is required to recover the correct positions of the rows so as to ensure the correctness of the computation.

BoostGCN [64] utilizes a centralized load balancing scheme to allocate the tasks for its FAMs so as to resolve load imbalance caused by uneven degree distribution. A task pool containing all 3-D partitions is maintained, and tasks are assigned to each FAM at 3-D partition granularity.

G-CoS [78] uses a flexible workload allocation including two flexible workload allocation schemes to ensure workload better fit sub-accelerators' micro-architecture. The processing element (PE) array's dimensions and tiling sizes can be adjusted to achieve high hardware utilization and efficiency, which balancing the workload with the number of feature rows and the number of weight columns respectively.

LW-GCN [79] employs pre-processing steps to balance workload for each pair of tiles during computation as it utilizes out-product tiling approach. Each PE is able to work independently and starts computation of a new row immediately when the previous one is finished, so as to increase PE efficiency. Multiple rows are effectively concatenated before assigned to PEs to eliminate idle time.

4. Processing-in-memory optimizations

Due to unique characteristics of emerging systems and devices, such as ReRAM-based crossbars for processing-in-memory, some of the optimization techniques and their objectives are tightly coupled with their implementations.

Both PIMGCN [27] and PASGCN [76] aim to reduce the hardware overhead of ReRAM crossbars induced by data mapping, and choose a design with CAM crossbars and MAC crossbars for GCN acceleration to eliminate the data duplication, lowering the overhead of crossbars to store extra data, so that the area occupation and energy consumption can be significantly reduced.

ReFlip [73] and ReaDy [74] adopt a flipped mapping to improve the utilization of crossbars and fully exploit intra-vertex parallelism between features (Figure 10), in which vertex features are mapped into crossbars and edge data are fed as input of crossbars.

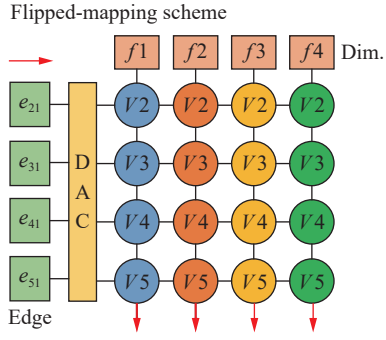

Figure 10 The flipped-mapping scheme in ReFlip [73]

ReGNN [61] designs three sub-engines for vertices of different degrees, and provides a vertex scheduler to assign different aggregation tasks to different sub-engines, so that the processing efficiency of the aggregation engine can be improved.

Ogbogu *et al.* [95] introduce a crossbar-aware pruning technique, termed DietGNN, targeted for training purpose to reduce crossbar overhead and energy consumption. Since selective shutdown of certain rows or columns of a crossbar may not contribute to energy reservation, DietGNN aims to shut an entire crossbar by pruning the data mapped to it. DietGNN first prunes the GNN model based on the crossbar knowledge to produce a wining ticket, i.e., an optimal sub-network, inspired by

state-of-the-art pruning techniques, such as the *lottery ticket hypothesis* [96], and then train the winning ticket on a ReRAM-based platform. DietGNN incorporates key characteristics of hardware and extracts a hardware-friendly sub-network from the original one that can be trained on a crossbar diet without compromising accuracy.

V. Discussions and Challenges

Hardware acceleration for GNNs has achieved great breakthroughs in efficient processing of GNNs, significantly outperforming typical software frameworks for GNNs on CPUs and GPUs in overall throughput, latency, and energy efficiency of the whole system. Though different from algorithmic GNNs defined and executed by conventional software frameworks, hardware acceleration for GNNs exactly offers GNNs defined by underlying hardware platform with customized execution dataflow without modifying the algorithmic representation of GNNs. In this section, we provide discussions on issues that may be concerned by researchers.

Extra overhead In fact, the existing hardware approaches can not work independently of software frameworks, since they either target at inference or training with limited capability. Generally, hardware approaches that target at inference of GNNs require trained weight matrices of each layer, the adjacency matrix of the graph data, and feature matrices of vertices or edges as the input sources of their architecture. Sometimes the adjacency matrix is processed by algorithmic methods, depending on specific requirements of GNNs. A little bit different from approaches targeting at inference, those targeting at training require layer-wise initialized weight matrices, the adjacency matrix, feature matrices of vertices or edges, and representation of specific necessary functions, e.g., graph sampling and dropout, as inputs of the architectures. Fortunately, all the inputs are consistent with those used in inference and training of algorithmic GNNs. They have to be computed only once, and then they can be shared by both algorithmic approaches and hardware approaches afterwards. In this way, the overhead of pre-processing is amortized. With respect to format conversion overhead, different from accelerators for basic operations such as SpMM and SpMV that have upstream tasks, the input sources of GNN accelerators have been transformed into correct formats only once before execution, and no format conversion is needed during the execution. Since the format conversion before the execution is not tracked in evaluation by existing approaches, the overhead of such a kind of pre-processing can be neglected. Thus, the hardware acceleration itself incurs little extra overhead in respect of the input sources. Some of the hardware approaches follow a software-hardware co-design methodology and introduce algorithmic methods in their system, but the algorithmic component and hardware component usually form a macro pipeline so that potential extra costs can be reduced as much as possible.

Versatility Existing hardware approaches of acceleration for GNNs support different variants of GNNs, as shown in Tables 2-4. Given a certain approach, its architecture is usually designed to cope with the computation of workloads with similar characteristics. Thus, the approaches designed for different variants may differ in the architecture since they target at different execution dataflow. Some of the hardware approaches may introduce special function units to extend the overall function of the system, such as GNNIE [72]. However, with respect to architecture-aware optimization techniques in Section IV, few of them are dedicated to certain GNN workloads, since they target at characteristics of the execution dataflow, which is related with intrinsic characteristics of graph-structured data and the architecture design. Hence, the architecture design and the customized dataflow may be dedicated to certain GNN workloads, but different approaches may share general optimization techniques.

Performance The existing hardware acceleration approaches choose different SOTA works as baselines and use various GNN applications to evaluate their hardware design. Most of them have achieve extremely high speedup and energy efficiency than typical software frameworks on CPUs and GPUs. However, the complexity of implementation and the workloads can increase the uncertainty of performance evaluation. Generally, SOTA accelerators are first re-implemented and then scaled to match the on-chip resources of the evaluation platform, due to difference between their original hardware platforms. Moreover, since most of the hardware acceleration approaches for GNNs are based on ASIC technology, and they mostly concerns about the area and power consumption of hardware components, the analysis of on-chip resource utilization are somewhat neglected, but that is critical to approaches based on FPGAs. Some approaches may have mentioned this issue in their papers, but their metrics are not unified. `GraphACT` [58] compares the utilization of DSP slices and BRAM while DNNs working on different graph dataset is computed, and HP-GNN [68] further analyzes the overall utilization of LUTs. Detailed analysis on the utilization of hardware components is needed and that can contribute to FPGA-based accelerators.

Pros and cons The three categories of hardware approaches have their strengths and weaknesses respectively, despite the effectiveness in evaluation on certain workloads. Hybrid architectures usually have high frequency to reduce processing latency and increase overall throughput, so as to maximize the performance of the pipelined execution dataflow. Optimizations are introduced separately to different compute engines. With proper scheduling strategies and pipeline control logic, hybrid architectures can be enhanced with a flexible execution dataflow to support different computation order of the different phases, leading to improvement of versatility. However, the on-chip buffers for intermediate re-

sults may occupy a considerable amount of total on-chip resources. The data transmission by read and write operations can incur extra latency and energy consumption, leading to degradation of the overall performance and energy efficiency. Moreover, hybrid architectures pose strict requirements on pipeline scheduling strategies, and poorly designed strategies can also reduce the overall efficiency. With respect to holistic architectures, they do not usually need on-chip buffers to store and transfer intermediate results, and all the intermediate results are kept with the compute engines during the execution. Thus, holistic architectures usually have respectively high energy efficiency, and homogeneous compute engines can reach high utilization with proper configuration. However, holistic architectures rely on methods to extract common operations from the computation to design homogeneous compute engines to compute them, and extra software and hardware components are required to handle exceptions, leading to complexity of the design. With respect to large-scale architectures, they are heavily driven by industrial demands. Those architectures are evaluated on limited number of GNN workloads, and their capability of processing other GNN variants remains to be studied. For those that adopts topological networks for data exchange between nodes, the communication can be a bottleneck of the whole system, and a dedicated efficient topology network is required for those architectures.

VI. Future Directions

Existing researches on hardware acceleration for GNNs have proved their efficiency and effectiveness for accelerating either training or inference phase of various workloads, i.e., variants of GNNs and graph-structured data with clearly different characteristics. They have outperformed general-purpose platforms and SOTA solutions significantly on several critical performance metrics. However, problems still lie somewhere, primarily due to the rapid development of approaches for deep learning on graph, and the evolution of the practical demands of industries, which can lead to increasing complexity of graph-structured data and variants of GNNs. In other words, current solutions, even SOTA ones, may not be able to cope with workloads that appears afterwards. Thus, we would give four suggestions about the future directions of hardware acceleration for GNNs to facilitate our following researches.

Acceleration for complicated variants of GNNs The fast evolution of algorithms for deep learning on graphs have born a number of variants of GNNs. Though most of them originate from the family of GCNs [2] and share some characteristics, they are quite different in some aspects. Typically, both GATs [4] and GINs [3] can be seen as a kind of GCN, but many of the hardware acceleration designed for GCNs are inadequately efficient to accelerate GINs and GATs, due to distinct functions adopted by the two kind of models. Moreover, both GraphSAGE-Pool [1] and STGNN [97] combines several computation kernels to improve capability of representation. Thus, GNNerator [60] adopts an execution pipeline that can be configured to support different GCN variants. GNNIE [72] introduces special processing units to execute the LeakyReLU activation functions of GATs, and it further optimize the workflow for GATs' aggregation. However, most of researches pay few attention to those variants, and optimization techniques for them are limited. To further improve the overall efficiency of acceleration, researches should take the variants into consideration.

Acceleration for large-scale GNNs The amount of graph-structured in industries is extraordinarily large-scale, which is almost impossible to store and process on a single machine. Large-scale distributed processing is the most commonly utilized solution to cope with large-scale graph learning. Meanwhile, in order to acquire exact knowledge from large-scale graph-structured data, the amount of GNNs adopted in practical scenarios can also be extremely large, which costs expensive time and energy overhead to complete model training and inference. Graphite [81] provide a hardware-assisted aggregation using a modified DMA engine to accelerate GNNs' execution on CPUs in datacenters. SmartSAGE [82] provides a solution for in-storage-processing GNN training. Li *et al.* [83] offers a practical solution for deploying a FaaS system for GNN acceleration in a typical datacenter. GNNear [84] aims to cope with full-batch training of large-scale graph deep learning which needs large amount of memory space. MultiGCN [85] targets at GCNs on large-scale graph-structured data. Most of researches on large-scale GNN acceleration are mainly conducted on general-purpose platforms with modifications applied to the original architecture, focusing on algorithm design to alleviate bandwidth- and latency-related problems. Thus, there remains works to be done if oriented to requirements of industries.

Acceleration with emerging devices Processing-in-memory architectures have been adopted as a choice to efficiently accelerate execution of GNNs. Emerging devices, such as ReRAM crossbars, have been used for the MVM computations. PIMGCN [27] offers a state-of-the-art PIM-based accelerator using ReRAM crossbars, and achieves breakthroughs on both speedup and energy efficiency. Though it achieves both high energy efficiency and high speedup over previously designed hybrid architectures, it does not achieve significant speedup over another SOTA holistic solution, i.e., AWB-GCN. Since there exist only a few researches based on processing-in-memory architectures and most of them targets at energy efficiency, future researches may adopt approaches to further elevate the throughput of PIM-based architectures to acquire higher speedup. Moreover, with other emerging devices beyond ReRAM crossbars, other novel architectures may be designed.

Algorithm-architecture co-design Algorithms and

software applications are easily to be developed and modified. Hardware acceleration for GNNs typically follows a software-hardware co-designed or algorithm-hardware co-designed methodology. Here, we would refer to algorithm-hardware co-design as implementing a certain algorithm with the underlying hardware to create a workflow consistent with it. Contrary to existing lightweight graph reordering algorithms, I-GCN [28] propose a hardware-assisted runtime graph reordering approach. The algorithms proposed by I-GCN are exactly the execution flow of processing elements and the overhead of reordering is eliminated by overlapping the stages of reordering and computing. Future researches can adopt such a methodology and propose algorithms of runtime versions to support efficient processing. Intuitively, a number of SOTA graph processing algorithms may also be modified into runtime ones to create efficient workflows.

Heuristic-hardware co-operation Heuristics hold prior knowledge about the environment, which can be utilized as useful guidelines for operations. Following software-hardware co-designed methodology, PASGCN [76] propose a lightweight GCN to learn knowledge about the input graph together with information of ReRAM crossbars in PIMGCN [27] to directly predict neighbors during inference, leading to extremely high energy efficiency and speedup over SOTA solutions. Ogbogu *et al.* [95] propose DietGNN which performs pruning based on information of the crossbars to achieve high energy efficiency during training stage. The application of heuristics can generate hardware-aware networks that can be executed efficiently on the underlying hardware, but existing works are limited to ReRAM crossbars. Future researches may develop more sophisticated heuristics to further improve the effectiveness of acceleration, or develop heuristic-based approaches that can be applied to more kinds of devices.

VII. Conclusion

In this article, we provide a survey on hardware acceleration for GNNs. We propose a methodology of categorization and classify existing researches into three categories based on the type of their hardware architectures: hybrid architectures, holistic architectures, and large-scale architectures. We introduce the overall design of representative approaches and recent advances. Then, we introduce their optimization techniques at different levels and conduct a concise analysis upon them. Finally, we propose five suggestions on future directions of hardware acceleration for GNNs.

Acknowledgements

This work was supported by the National Natural Science Foundation of China Key Program (Grant No. 62032001) and General Program (Grant No. 61972407).

References

[1] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Long Beach, CA, USA, pp. 1025–1035, 2017.

[2] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proceedings of the 5th International Conference on Learning Representations*, Toulon, France, 2017.

[3] K. Xu, W. H. Hu, J. Leskovec, *et al.*, "How powerful are graph neural networks?," in *Proceedings of the 7th International Conference on Learning Representations*, New Orleans, LA, USA, 2019.

[4] P. Veličković, G. Cucurull, A. Casanova, *et al.*, "Graph attention networks," in *Proceedings of the 6th International Conference on Learning Representations*, Vancouver, Canada, 2018.

[5] R. Ying, D. Bourgeois, J. X. You, *et al.*, "GNNExplainer: Generating explanations for graph neural networks," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Vancouver, Canada, article no. 829, 2019.

[6] H. T. Nguyen, Q. D. Ngo, and V. H. Le, "IoT botnet detection approach based on PSI graph and DGCNN classifier," in *Proceedings of 2018 IEEE International Conference on Information Communication and Signal Processing (ICICSP)*, Singapore, pp. 118–122, 2018.

[7] R. Zhu, K. Zhao, H. X. Yang, *et al.*, "AliGraph: A comprehensive graph neural network platform," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2094–2105, 2019.

[8] T. Xie and J. C. Grossman, "Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties," *Physical Review Letters*, vol. 120, no. 14, article no. 145301, 2018.

[9] M. Zitnik, M. Agrawal, and J. Leskovec, "Modeling polypharmacy side effects with graph convolutional networks," *Bioinformatics*, vol. 34, no. 13, pp. i457–i466, 2018.

[10] C. W. Coley, W. G. Jin, L. Rogers, *et al.*, "A graph-convolutional neural network model for the prediction of chemical reactivity," *Chemical Science*, vol. 10, no. 2, pp. 370–377, 2019.

[11] Y. X. Liu, N. Zhang, D. Wu, *et al.*, "Guiding cascading failure search with interpretable graph convolutional network," *arXiv preprint*, arXiv: 2001.11553, 2020.

[12] J. Chen, T. F. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," in *Proceedings of the 6th International Conference on Learning Representations*, Vancouver, Canada, 2018.

[13] A. Bojchevski, J. Gasteiger, B. Perozzi, *et al.*, "Scaling graph neural networks with approximate PageRank," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Virtual Event, CA, USA, pp. 2464–2473, 2020.

[14] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric," *arXiv preprint*, arXiv: 1903.02428, 2019.

[15] M. J. Wang, D. Zheng, Z. H. Ye, *et al.*, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint*, arXiv: 1909.01315, 2019.

[16] Z. Q. Lin, C. Li, Y. S. Miao, *et al.*, "PaGraph: Scaling GNN training on large graphs via computation-aware caching," in *Proceedings of the 11th ACM Symposium on Cloud Computing*,Virtual Event, USA, pp. 401–415, 2020.

[17] Q. X. Sun, Y. Liu, H. L. Yang, *et al.*, "CoGNN: Efficient scheduling for concurrent GNN training on GPUs," in *Proceedings of SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, TX, USA, pp. 1–15, 2022.

[18] Y. T. Gui, Y. D. Wu, H. Yang, *et al.*, "HGL: Accelerating heterogeneous GNN training with holistic representation and

optimization," in *Proceedings of SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, TX, USA, pp. 1–15, 2022.

[19] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *Proceedings of 2020 57th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, pp. 1–6, 2020.

[20] M. Y. Yan, Z. D. Chen, L. Deng, *et al.*, "Characterizing and understanding GCNs on GPU," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 22–25, 2020.

[21] M. Y. Yan, L. Deng, X. Hu, *et al.*, "HyGCN: A GCN accelerator with hybrid architecture," in *Proceedings of 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, San Diego, CA, USA, pp. 15–29, 2020.

[22] S. Q. Zhang, Z. Qin, Y. H. Yang, *et al.*, "Transparent partial page migration between CPU and GPU," *Frontiers of Computer Science*, vol. 14, no. 3, article no. 143101, 2020.

[23] S. J. Fan, J. W. Fei, and L. Shen, "Accelerating deep learning with a parallel mechanism using CPU + MIC," *International Journal of Parallel Programming*, vol. 46, no. 4, pp. 660–673, 2018.

[24] T. Geng, A. Li, R. B. Shi, *et al.*, "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Athens, Greece, pp. 922–936, 2020.

[25] S. W. Liang, Y. Wang, C. Liu, *et al.*, "EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Transactions on Computers*, vol. 70, no. 9, pp. 1511–1525, 2021.

[26] J. J. Li, A. Louri, A. Karanth, *et al.*, "GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *Proceedings of 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Seoul, Korea, pp. 775–788, 2021.

[27] T. Yang, D. Y. Li, Y. B. Han, *et al.*, "PIMGCN: A ReRAM-based PIM design for graph convolutional network acceleration," in *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, pp. 583–588, 2021.

[28] T. Geng, C. S. Wu, Y. A. Zhang, *et al.*, "I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *Proceedings of MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, Virtual Event, Greece, pp. 1051–1063, 2021.

[29] H. R. You, T. Geng, Y. A. Zhang, *et al.*, "GCoD: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design," in *Proceedings of 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Seoul, Korea, pp. 460–474, 2022.

[30] X. Liu, M. Y. Yan, L. Deng, *et al.*, "Survey on graph neural network acceleration: An algorithmic perspective," in *Proceedings of the 31st International Joint Conference on Artificial Intelligence,* Vienna, Austria, pp. 5521–5529, 2022.

[31] S. Abadal, A. Jain, R. Guirado, *et al.*, "Computing graph neural networks: A survey from algorithms to accelerators," *ACM Computing Surveys*, vol. 54, no. 9, article no. 191, 2021.

[32] Z. H. Wu, S. R. Pan, F. W. Chen, *et al.*, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2021.

[33] Z. Y. Liu and J. Zhou, "Graph convolutional networks," in *Introduction to Graph Neural Networks*, Z. Y. Liu, J. Zhou, Eds. Springer, Cham, Switzerland, pp. 23–32, 2020.

[34] Z. Y. Liu and J. Zhou, "Graph recurrent networks," in *Introduction to Graph Neural Networks*, Z. Y. Liu, J. Zhou, Eds. Springer, Cham, Switzerland, pp. 33–37, 2020.

[35] Z. T. Liu and J. Zhou, "Graph attention networks," in *Introduction to Graph Neural Networks*, Z. Y. Liu, J. Zhou, Eds. Springer, Cham, Switzerland, pp. 39–41, 2020.

[36] K. Cho, B. van Merrienboer, C. Gulcehre, *et al.*, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar, pp. 1724–1734, 2014.

[37] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[38] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks,* Montreal, Canada, pp. 729–734, 200.

[39] L. Ruiz, F. Gama, and A. Ribeiro, "Gated graph recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 68, pp. 6303–6318, 2020.

[40] R. Ying, J. X. You, C. Morris, *et al.*, "Hierarchical graph representation learning with differentiable pooling," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, Montréal, Canada, pp. 4805–4815, 2018.

[41] J. Gilmer, S. S. Schoenholz, P. F. Riley, *et al.*, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning*, Sydney, Australia, pp. 1263–1272, 2017.

[42] Z. D. Chen, X. S. Li, and J. Bruna, "Supervised community detection with line graph neural networks," in *Proceedings of the 7th International Conference on Learning Representations*, New Orleans, LA, USA, 2019.

[43] G. H. Li, C. X. Xiong, A. Thabet, *et al.*, "DeeperGCN: All you need to train deeper GCNs," *arXiv preprint*, arXiv: 2006.07739, 2020.

[44] X. Bresson and T. Laurent, "Residual gated graph ConvNets," *arXiv preprint*, arXiv: 1711.07553, 2017.

[45] M. Schlichtkrull, T. N. Kipf, P. Bloem, *et al.*, "Modeling relational data with graph convolutional networks," in *Proceedings of the 15th European Semantic Web Conference*, Heraklion, Crete, Greece, pp. 593–607, 2018.

[46] Y. Wang, Y. B. Sun, Z. W. Liu, *et al.*, "Dynamic graph CNN for learning on point clouds," *ACM Transactions on Graphics*, vol. 38, no. 5, article no. 146, 2017.

[47] Y. N. Dauphin, A. Fan, M. Auli, *et al.*, "Language modeling with gated convolutional networks," in *Proceedings of the 34th International Conference on Machine Learning*, Sydney, Australia, pp. 933–941, 2017.

[48] J. X. You, R. Ying, X. Ren, *et al.*, "GraphRNN: Generating realistic graphs with deep auto-regressive models," in *Proceedings of the 35th International Conference on Machine Learning*, Stockholm, Sweden, pp. 5694–5703, 2018.

[49] Y. J. Li, O. Vinyals, C. Dyer, *et al.*, "Learning deep generative models of graphs," *arXiv preprint*, arXiv: 1803.03324, 2018.

[50] H. Y. Gao and S. W. Ji, "Graph u-nets," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 9, pp. 4948–4960, 2022.

[51] L. Zhao, Y. J. Song, C. Zhang, *et al.*, "T-GCN: A temporal graph convolutional network for traffic prediction," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 9, pp. 3848–3858, 2020.

[52] G. Panagopoulos, G. Nikolentzos, and M. Vazirgiannis, "Transfer graph neural networks for pandemic forecasting," in *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, Virtual Event, USA, pp. 4838–4845, 2021.

[53] H. Y. Gao, Z. Y. Wang, and S. W. Ji, "Large-scale learnable graph convolutional networks," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, London, UK, pp. 1416–1424, 2018.

[54] W. L. Chiang, X. Q. Liu, S. Si, *et al.*, "Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Anchorage, AK, USA, pp. 257–266, 2019.

[55] P. S. Huang, X. D. He, J. F. Gao, *et al.*, "Learning deep structured semantic models for web search using clickthrough data," in *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, San Francisco, CA, USA, pp. 2333–2338, 2013.

[56] N. P. Jouppi, C. Young, N. Patil, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, Toronto, Canada, pp. 1–12, 2017.

[57] B. Y. Zhang, H. Q. Zeng, and V. Prasanna, "Hardware acceleration of large scale GCN inference," in *Proceedings of IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Manchester, UK, pp. 61–68, 2020.

[58] H. Q. Zeng and V. Prasanna, "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Seaside, CA, USA, pp. 255–265, 2020.

[59] C. Chen, K. L. Li, X. F. Zou, and Y. F. Li, "DyGNN: Algorithm and architecture support of dynamic pruning for graph neural networks," in *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, pp. 1201–1206, 2021.

[60] J. R. Stevens, D. Das, S. Avancha, *et al.*, "GNNerator: A hardware/software framework for accelerating graph neural networks," in *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, pp. 955–960, 2021.

[61] C. Liu, H. K. Liu, H. Jin, *et al.*, "ReGNN: A ReRAM-based heterogeneous architecture for general graph neural networks," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, San Francisco, CA, USA, pp. 469–474, 2022.

[62] J. X. Chen, Y. Q. Lin, K. Y. Sun, *et al.*, "GCIM: Toward efficient processing of graph convolutional networks in 3d-stacked memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 3579–3590, 2022.

[63] C. Chen, K. L. Li, Y. F. Li, *et al.*, "ReGNN: A redundancy-eliminated graph neural networks accelerator," in *Proceedings of 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Seoul, Korea, pp. 429–443, 2022.

[64] B. Y. Zhang, R. Kannan, and V. Prasanna, "BoostGCN: A framework for optimizing GCN inference on FPGA," in *Proceedings of 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Orlando, FL, USA, pp. 29–39, 2021.

[65] S. W. Liang, C. Liu, Y. Wang, *et al.*, "DeepBurning-GL: An automated framework for generating graph neural network accelerators," in *Proceedings of the 2020 IEEE/ACM International Conference on Computer Aided Design*, San Diego, CA, USA, pp. 1–9, 2020.

[66] Y. Zhu, Z. H. Zhu, G. H. Dai, *et al.*, "Exploiting parallelism with vertex-clustering in processing-in-memory-based GCN accelerators," in *Proceedings of 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Antwerp, Belgium, pp. 652–657, 2022.

[67] C. M. Zhang, T. Geng, A. Q. Guo, *et al.*, "H-GCN: A graph convolutional network accelerator on versal ACAP architecture," in *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, Belfast, UK, pp. 200–208, 2022.

[68] Y. C. Lin, B. Y. Zhang, and V. Prasanna, "HP-GNN: Generating high throughput GNN training implementation on CPU-FPGA heterogeneous platform," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 123–133, 2022.

[69] W. T. Hou, K. Zhong, S. L. Zeng, *et al.*, "NTGAT: A graph attention network accelerator with runtime node tailoring," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, Tokyo, Japan, pp. 1–6, 2023.

[70] X. K. Song, T. Zhi, Z. Fan, *et al.*, "Cambricon-G: A polyvalent energy-efficient accelerator for dynamic graph neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 1, pp. 116–128, 2022.

[71] J. J. Li, H. Zheng, K. Wang, *et al.*, "SGCNAX: A scalable graph convolutional neural network accelerator with workload balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2834–2845, 2022.

[72] S. Mondal, S. D. Manasi, K. Kunal, *et al.*, "GNNIE: GNN inference engine with load-balancing and graph-specific caching," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, San Francisco, CA, USA, pp. 565–570, 2022.

[73] Y. Huang, L. Zheng, P. C. Yao, *et al.*, "Accelerating graph convolutional networks using crossbar-based processing-in-memory architectures," in *Proceedings of 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Seoul, Korea, pp. 1029–1042, 2022.

[74] Y. Huang, L. Zheng, P. C. Yao, *et al.*, "ReaDy: A ReRAM-based processing-in-memory accelerator for dynamic graph convolutional networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 3567–3578, 2022.

[75] X. B. Chen, Y. K. Wang, X. F. Xie, *et al.*, "Rubik: A hierarchical architecture for efficient graph neural network training," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 4, pp. 936–949, 2022.

[76] T. Yang, D. Y. Li, F. Ma, *et al.*, "PASGCN: An ReRAM-based PIM design for GCN with adaptively sparsified graphs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 1, pp. 150–163, 2023.

[77] C. Peltekis, D. Filippas, C. Nicopoulos, *et al.*, "FusedGCN: A systolic three-matrix multiplication architecture for graph convolutional networks," in *Proceedings of IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Gothenburg, Sweden, pp. 93–97, 2022.

[78] Y. A. Zhang, H. R. You, Y. G. Fu, *et al.*, "G-CoS: GNN-accelerator co-search towards both better accuracy and efficiency," in *Proceedings of 2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, Munich, Germany, pp. 1–9, 2021.

[79] Z. F. Tao, C. Wu, Y. Liang, *et al.*, "LW-GCN: A lightweight FPGA-based graph convolutional network accelerator," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 1, article no. 10, 2022.

[80] A. I. Arka, B. K. Joardar, J. R. Doppa, *et al.*, "DARE: Droplayer-aware manycore ReRAM architecture for training graph neural networks," in *Proceedings of 2021 IEEE/ACM*

International Conference On Computer Aided Design (ICCAD), Munich, Germany, pp. 1–9, 2021.

[81] Z. X. W. Gong, H. X. Ji, Y. Yao, *et al.*, "Graphite: Optimizing graph neural networks on CPUs through cooperative software-hardware techniques," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, New York, NY, USA, pp. 916–931, 2022.

[82] Y. Lee, J. Chung, and M. Rhu, "SmartSAGE: Training large-scale graph neural networks using in-storage processing architectures," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, New York, NY, USA, pp. 932–945, 2022.

[83] S. C. Li, D. M. Niu, Y. H. Wang, *et al.*, "Hyperscale FPGA-as-a-service architecture for large-scale distributed graph neural network," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, New York, NY, USA, pp. 946–961, 2022.

[84] Z. Zhou, C. Li, X. C. Wei, *et al.*, "GNNear: Accelerating full-batch training of graph neural networks with near-memory processing," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Chicago, Illinois, pp. 54–68, 2022.

[85] G. J. Sun, M. Y. Yan, D. Wang, *et al.*, "Multi-node acceleration for large-scale GCNs," *IEEE Transactions on Computers*, vol. 71, no. 12, pp. 3140–3152, 2022.

[86] B. Gaide, D. Gaitonde, C. Ravishankar, *et al.*, "Xilinx adaptive compute acceleration platform: Versal^{TM} architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Seaside, CA, USA, pp. 84–93, 2019.

[87] F. Manessi, A. Rozza, and M. Manzo, "Dynamic graph convolutional networks," *Pattern Recognition*, vol. 97, article no. 107000, 2020.

[88] N. Challapalle, S. Rampalli, L. H. Song, *et al.*, "GaaS-X: Graph analytics accelerator supporting sparse data representation using crossbar architectures," in *Proceedings of ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, Valencia, Spain, pp. 433–445, 2020.

[89] P. Chi, S. C. Li, C. Xu, *et al.*, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proceedings of ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Seoul, Korea, pp. 27–39, 2016.

[90] Y. Rong, W. B. Huang, T. Y. Xu, *et al.*, "DropEdge: Towards deep graph convolutional networks on node classification," in *Proceedings of the 8th International Conference on Learning Representations*, Addis Ababa, Ethiopia, pp. 1-17, 2019.

[91] N. Srivastava, G. Hinton, A. Krizhevsky, *et al.*, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[92] N. P. Jouppi, D. H. Yoon, G. Kurian, *et al.*, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.

[93] Y. M. Zhang, V. Kiriansky, C. Mendis, *et al.*, "Making caches work for graph analytics," in *Proceedings of 2017 IEEE International Conference on Big Data (Big Data)*, Boston, MA, USA, pp. 293–302, 2017.

[94] X. W. Zhu, W. T. Han, and W. G. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, Santa Clara, CA, USA, pp. 375–386, 2015.

[95] C. Ogbogu, A. I. Arka, B. K. Joardar, *et al.*, "Accelerating large-scale graph neural network training on crossbar diet," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 3626–3637, 2022.

[96] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *Proceedings of the 7th International Conference on Learning Representations*, New Orleans, LA, USA, pp. 1-42, 2019.

[97] X. Y. Wang, Y. Ma, Y. Q. Wang, *et al.*, "Traffic flow prediction via spatial temporal graph neural network," in *Proceedings of the Web Conference 2020*, Taipei, China, pp. 1082–1092, 2020.

Shi CHEN received the B.E. degree in computer science and technology from National University of Defense Technology, Changsha, China, in 2017. He is a Ph.D. candidate in the College of Computer, National University of Defense Technology, Changsha, China. His research interests include computer architecture and graph-based hardware accelerator.

(Email: chenshi17@nudt.edu.cn)

Jingyu LIU received the M.S. degree in integrated circuit engineering from National University of Defense Technology, Changsha, China, in 2021. He is a Ph.D. candidate in the College of Computer, National University of Defense Technology, Changsha, China. His research interests include computer architecture, SoC designs, and microprocessor architecture.

(Email: liujingyu@nudt.edu.cn)

Li SHEN received the B.S., M.S., and Ph.D. degrees in computer science and technology from National University of Defense Technology, Changsha, China. He is a Professor at College of Computer, National University of Defense Technology, Changsha, China. His research interests include high performance processor architecture, parallel programming, and performance optimization techniques.

(Email: lishen@nudt.edu.cn)