Journal of Big Data

SURVEY **Open Access**

A survey of graph convolutional networks (GCNs) in FPGA-based accelerators

Marco Procaccini[1,3,4*], Amin Sahebi[1,2] and Roberto Giorgi[1,3*]

*Correspondence:
marco.procaccini@cnr.it;
giorgi@unisi.it

[1] Department of Information Engineering and Mathematics, University of Siena, Siena, Italy
[2] Department of Information Engineering, University of Florence, Florence, Italy
[3] Consorzio Interuniversitario Nazionale per l'Informatica, Rome, Italy
[4] National Research Council, Pisa, Italy

Abstract

This survey overviews recent Graph Convolutional Networks (GCN) advancements, highlighting their growing significance across various tasks and applications. It underscores the need for efficient hardware architectures to support the widespread adoption and development of GCNs, particularly focusing on platforms like FPGAs known for their performance and energy efficiency. This survey also outlines the challenges in deploying GCNs on hardware accelerators and discusses recent efforts to enhance efficiency. It encompasses a detailed review of the mathematical background of GCNs behind inference and training, a comprehensive review of recent works and architectures, and a discussion on performance considerations and future directions.

Keywords: Graph Convolutional Networks, Hardware acceleration, FPGA, Heterogeneous platform

Introduction

One of the most notable recent advancements of Deep Learning lies in the successful implementation of Graph Convolutional Networks (GCNs) [1–19]. The central idea behind GCNs [20] is the iterative aggregation of feature data from graph's nodes neighborhoods using neural networks, as shown in Fig. 1. With a single "convolution-like" operation, feature data is transformed and collected from a node's immediate neighborhood in the graph. By stacking multiple convolution layers, the flow of information can reach distant areas of the graph. Unlike deep models that rely solely on the content (e.g., recurrent neural networks [21]), GCNs use both the content data and the graph structure.

GCNs are demonstrating significant promise across a range of tasks [22–25]. Notably, major companies like Alibaba, Facebook, and Google have implemented GCNs in their data centers, underscoring the expanding significance and potential applications of this technology [26]. Given this growing landscape, there is a need for an efficient and high-performance architecture for solving critical real-life problems and fostering more research on GCNs. Thus, based on the aforementioned context, we articulate our motivation for surveying GCN implementations, in particular on platforms able to achieve good performance while keeping a good energy efficiency, such as the FPGAs [1].

GCNs extend the capabilities of Graph Neural Networks (GNNs) by incorporating graph convolutional operations inspired by traditional convolutional layers in convolutional neural networks (CNNs). By employing graph convolutions, GCNs are capable of effectively extracting hierarchical features from graph-structured data, allowing for machine-learning tasks such as node classification, link prediction, and graph-level prediction. GCN algorithms generally incorporate processing nodes (vertices) and their connections (edges) in a graph, where each node has associated features. However, deploying GCNs efficiently on hardware accelerators like FPGAs poses challenges. One challenge is the heavy memory usage and irregular access patterns during the feature aggregation phase, in which the information is gather from the neighbors of a given node. Another issue is the uneven distribution of workloads among nodes, which can lead to inefficient resource utilization [27, 28]. Additionally, within each layer of a GCN, there's an imbalance in the workload between the memory-intensive aggregation phase and the computation-heavy transformation phase, which can slow down the process. While GCNs offer powerful capabilities for analyzing graph-structured data, optimizing their performance on hardware accelerators like FPGAs requires overcoming technical obstacles. To tackle these challenges, there are recent studies to try improving efficiency in memory usage, workload distribution, and computational tasks [29–33].

This survey reviews the fundamental background of GCNs, covered in the problem statement in Section "Problem definition: background and mathematical foundations". It organizes and discusses recent works in the field, presented in Section Overview of existing solutions. The survey also reviews the most effective architectures for GCNs, detailed in Section Proposed Solutions: Architecture Highlights of Selected FPGA-based GCNs. Finally, it explores the performance-cost tradeoffs and highlights promising future directions in Section Elaboration.

Problem definition: background and mathematical foundations

GCNs aim to improve accuracy in learning arbitrary graph-structured data. The basic idea behind GCNs is applying a method for capturing nearby information similarly to Convolutional Neural Networks (CNNs) [34, 35] directly on a matrix representing the graph and the feature embeddings associated with each graph's node.

The initial proposal was used in the semi-supervised classification of graph-structured data [20]. The main goal is to select the information embedded within the graph more accurately than with previous techniques.

The data are features that are represented by, e.g., a vector $\mathbf{x} \in \mathbb{R}^c$ which is named *embedding*, associated with a node or edge of the graph, where c is the number of elements of the embedding or features in the inputs. The n embeddings can be organized in a matrix $\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}]^T \in \mathbb{R}^{n \times c}$ and the GCN produces a predicted set of f output classes or output labels as a matrix $\hat{\mathbf{Y}} \in \mathbb{R}^{n \times f}$ so that:

$$\hat{\mathbf{Y}} = GCN(\mathbf{A}, \mathbf{X}) \tag{1}$$

where $\mathbf{A}$ is a graph description, such as its adjacency matrix. In this way, GCNs can be naturally deployed in Machine Learning to address several classical tasks, such as:

- *Node Classification* For graphs where features and labels characterize nodes, the objective is to forecast the label of each node. For instance, in a social network scenario, nodes may represent users, and the aim could involve predicting user attributes or interests.
- *Link Prediction* Given a graph, the goal is to anticipate either if an edge is present between two nodes or the probability of a potential edge forming between them. This task is often relevant in recommendation systems, where predicting connections between users or items can improve personalized recommendations.
- *Graph Classification* For a collection of graphs, one may want to classify or characterize entire graphs. For example, in molecular biology, graphs may represent chemical compounds, and one may desire to predict the properties of the molecules or classify them into different classes based on their structure.
- *Community Detection* Given a graph, identify communities or node clusters that are densely connected internally but sparsely connected with nodes in other communities. Community detection is relevant in various fields, e.g., social network analysis, biology, and computer networks.
- *Graph Generation* Produce new graphs with structural properties similar to a given set of graphs. Graph generation has generative modeling, drug discovery, and network synthesis applications.

Why GCNs attracted attention

The combination of flexibility [36], scalability [37], interpretability [38], and effectiveness [39] in learning from graph-structured data has made GCNs a prominent area of research and a powerful tool to address real-world problems in diverse domains [26].

Li et al. [40] showed how GCN could outperform Fully-Connected Networks (FCNs) in terms of prediction accuracy, as demonstrated by the higher accuracy of a two-layers GCN in Table 1. The advantage of GCN is quite clear. The only difference between the layer-wise propagation rule of the FCNs and the GCNs is the consideration of the graph convolution matrix $\mathbf{G}_\theta$ (see Equation 3), which only relates to the adjacency matrix, as it is shown in the following.

In the latest decade, GCN concepts and variants have been intensively developed and surveyed [26, 36, 41–43], following their promising performance and relatively simple implementation.

More in detail, the work of Ju et al. [26] provides a comprehensive survey of current deep graph representation learning algorithms, categorizing them by neural network

Table 1 GCNs vs. Fully-Connected Networks [40] while performing the task of a semi-supervised classification (dataset: Cora citation network; labels: 20 for each class)

Model	FCN One-layer	FCN Two-layers	GCN One-layer	GCN Two-layers
Accuracy	0.530860	0.559260	0.707940	**0.798361**

The bold value indicates the superior accuracy of the two-layers GCN over the Fully-Connected Networks (FCN) architectures and advanced learning paradigms, including GCNs. However the analysis is only focusing on the mathematical aspects, not the computer architecture.

Li et al. [36] review FPGA-based accelerators for graph convolutional networks (GCNs) until 2022, detailing challenges, design solutions, performance metrics, and future directions for improving algorithm-hardware co-design, scheduling efficiency, capacity of accommodating different algorithms, and speed of development.

However, this survey lacks to highlight the mathematical background necessary for inference and for training, which is hardly found in a single point as we offer in this survey, for an in-depth understanding of the possibilities in the FPGA implementations. Moreover, we added more recent works and a more extensive direct comparison of the architectural implementations.

Garg et al. [42] present a taxonomy for describing and comparing the diverse dataflow and microarchitecture optimizations used in custom accelerators for Graph Neural Networks (GNNs), addressing the unique compute and memory features of GNNs compared to Deep Neural Networks (DNNs), but the review is more tailored to GPUs rather than to FPGAs.

The survey of Abadal et al. [44] reviews the area of Graph Neural Networks (GNNs) from a computing perspective, summarizing GNN fundamentals, algorithmic evolution, and operational phases, while providing a detailed analysis of recent software and hardware for accelerating computations, and proposing graph-aware solutions for GNN accelerators based on hardware-software codesign. However, there is no discussion of FPGA-based architecture and the mathematical background discussion is limited.

The survey of Zhou et al. [41] proposes a general design pipeline for Graph Neural Network (GNN) models in which GCN are only a limited part, discusses the variants of each component, indicates software-only implementations, categorizes their applications, and identifies some open problems for future research in the more general area of GNNs.

In the following Table 2, we highlight the major differences related to the focus of this survey and the other mentioned surveys.

Mathematical foundations of Graph Convolutional Networks (GCNs)

The mathematical foundations of GCNs originate from aggregating local information via the same filtering technique used in CNNs and DSP [45]. This idea works in practice quite well, although there is no direct mathematical derivation from CNNs: it can

Table 2 Major highlights of this survey compared to other works that survey GCNs

FEATURE	Our Work	Ju et al. [26]	Li et al. [36]	Garg et al. [42]	Abadal et al. [44]	Zhou et al. [41]
FPGA latency comparison (no. of works compared)	19	7	0	0	0	0
FPGA architecture discussion	Y	Y	N	N	N	N
Inference and training detailed math overview of classic GCNs	I+T	N	I	N	Limited	I

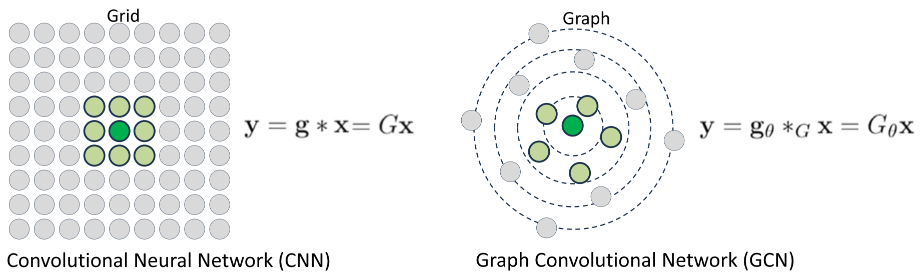
Grid
$y = g * x = Gx$
Convolutional Neural Network (CNN)
Graph
$y = g_\theta *_G x = G_\theta x$
Graph Convolutional Network (GCN)

Fig. 1 Conceptual operation of CNNs and GCNs (adapted from [47]). The GCNs aim to capture neighbor node features, similar to CNNs that capture, e.g., nearby pixels of an image

be seen as an example of the "Crossdisciplinarization" method for generating ideas (method "C" in [46]), in which something well-known is the Digital Signal Processing domain (DSP). i.e., the convolution filtering effect is applied to another well-known domain in Computer Science, i.e., graph theory, to solve Machine Learning tasks like classification (Fig. 1).

Graph convolution

In DSP, filtering a signal is obtained in the time domain via a convolution operation of an input signal with the filter's representation. In the discrete case, given two signals $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, a discrete filter $\mathbf{g} \in \mathbb{R}^n$ and the Fourier basis $[\phi_0, \phi_1, \ldots, \phi_{n-1}] := \Phi$ where[1]. $\Phi_{ij} := \frac{1}{\sqrt{n}} e^{\frac{2\pi jik}{n}}$, we can express the Discrete Fourier Transform (DFT) as $\hat{\mathbf{x}} := DFT(\mathbf{x}) = \Phi^T \mathbf{x}$, $\mathbf{y} := IDFT(\hat{\mathbf{y}}) = \Phi \hat{\mathbf{y}}, \hat{\mathbf{g}} := DFT(\mathbf{g}) = \Phi^T \mathbf{g}$, $\hat{\mathbf{G}} = diag(\hat{\mathbf{g}}) = diag(\hat{g}_0, \ldots, \hat{g}_{n-1})$ and the convolution operation[2] as:

$$\mathbf{y} = \mathbf{g} * \mathbf{x} = \Phi \hat{\mathbf{y}} = \Phi(\hat{\mathbf{g}} \odot \hat{\mathbf{x}}) = \Phi \hat{\mathbf{G}} \hat{\mathbf{x}} = \Phi \hat{\mathbf{G}} \Phi^T \mathbf{x} = \mathbf{G}\mathbf{x} \quad \text{where} \quad \mathbf{G} := \Phi \hat{\mathbf{G}} \Phi^T \tag{2}$$

Similarly, we can introduce the concept of *graph convolution* $*_\mathbf{G}$ with

$$\mathbf{y} = \mathbf{g}_\theta *_\mathbf{G} \mathbf{x} = \mathbf{G}_\theta \mathbf{x} \tag{3}$$

Where $\mathbf{G}_\theta$ can be seen as a function of a well-defined matrix that characterizes the graph, such as the *normalized graph Laplacian* $\mathbf{L}$ or an appropriate smoothing of it [40, 48]. In the landmark GCN work of Kipf and Welling [20], $\mathbf{G}_\theta = \hat{\mathbf{A}}$ has the role of aggregating information from the neighborhood nodes, as outlined in the following.

Derivation of the aggregation matrix $\hat{\mathbf{A}}$ *from the graph structure*

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges, $n = |\mathcal{V}|$ is the number of nodes in the graph, the (combinatorial) *graph Laplacian* matrix $\mathbf{L}_0$ is defined as $\mathbf{L}_0 := \mathbf{D} - \mathbf{A}$, where $\mathbf{A}$ is the adjacency matrix of the graph and $\mathbf{D}$ is the degree matrix of $\mathbf{G}$. The adjacency matrix describes the connections of the graph so that $\mathbf{A}_{ij} = 1$ if there is an edge between node v_i and v_j and $\mathbf{A}_{ij} = 0$ otherwise[3], where $v_i, v_j \in \mathbb{R}^n$ and $\mathbf{L}_0, \mathbf{D}, \mathbf{A} \in \mathbb{R}^{n \times n}$. The *degree matrix* $\mathbf{D} := diag(d_0, \ldots, d_{n-1})$ is a

[1] $\Phi := F^*$ where $F :=$ the Fourier matrix.
[2] The symbol $\odot$ denotes the Hadamard product.

[3] Here, for the sake of simplicity, we assume *undirected graphs*, i.e., all edges are bidirectional, but generalizations to directed edges exist.

diagonal matrix where $d_i = \sum_{j=0}^{n-1} \mathbf{A}_{ij}$ expresses the number of connections of each node v_i. The combinatorial Laplacian may contain big numbers in case of large graphs, so for numerical processing, it is preferred to use the *normalized graph Laplacian* $\mathbf{L} := \mathbf{D}^{-1/2}\mathbf{L}_0\mathbf{D}^{-1/2} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ instead.

In GCN processing, another important concept is aggregating information from neighbor nodes. The adjacency matrix naturally does this, expressing for each row (or column) the 1-hop-far connections to each node. However, it is normally important to process the embedding related to each node along with its neighbor embeddings: an implicit self-loop is added to that purpose. In terms of mathematical notation, this is expressed by using $\tilde{\mathbf{A}} := \mathbf{A} + \mathbf{I}$ as an adjacency matrix, which induces a degree matrix $\tilde{\mathbf{D}} := \mathbf{D} + \mathbf{I}$. By using $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{D}}$, Kipf and Welling [20] propose to use as aggregation matrix $\hat{\mathbf{A}} := \tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$ which can be seen as a “smoothed” Laplacian [48]. In other words, in Equation 3, the classical GCN assumes $\mathbf{G}_\theta = \hat{\mathbf{A}}$.

To derive this representation of the aggregation function $\mathbf{G}_\theta = \hat{\mathbf{A}}$, Kipf and Welling [20] make the following considerations. $\mathbf{A}$ contains only zeros and ones with an all-zeros diagonal. It may not permit its direct utilization as $\mathbf{G}_\theta$ in the graph convolution since numerical convergence may not be possible when applying $\mathbf{A}$ multiple times. Therefore, the Laplacian $\mathbf{L}$ is preferred since (for undirected graphs) $\mathbf{L}$ is a symmetric matrix (i.e., $\mathbf{L}^T = \mathbf{L}$) and also definite positive (i.e., $\mathbf{x}^T\mathbf{L}\mathbf{x} > 0, \forall \mathbf{x}$). Therefore, an orthogonal diagonalization always exists such that $\mathbf{L} = \mathbf{Q}\Lambda_L\mathbf{Q}^T$. With this in mind, we can see $\mathbf{G}_\theta$ as a function $f(\mathbf{L})$ and approximate $f()$ with the Chebyshev Polynomial Series (CPS)[4] such that $f(\mathbf{L}) = \mathbf{Q}f(\Lambda_L)\mathbf{Q}^T$. But $\mathbf{L}$ (and Λ_L) are known [45] to have eigenvalues in [0, 2], therefore we use $\tilde{f}(\tilde{\Lambda}_L)$ with $\tilde{\Lambda}_L = \Lambda_L - \mathbf{I}$ to shift the eigenvalues into [− 1,1] and apply the CPS approximation $f(\Lambda_L) = \tilde{f}(\tilde{\Lambda}_L) = \sum_{k=0}^{t} \theta_k T_k(\tilde{\Lambda}_L) = \sum_{k=0}^{t} \theta'_k \tilde{\Lambda}_L^k$. With the truncation for $t = 1$ to avoid involving farther nodes[5], and choosing $\theta'_0 = -\theta'_1 = \theta$ to reduce the number of (multiplication) operations, we obtain $f(\Lambda_L) \approx \theta(\mathbf{I} - \tilde{\Lambda}_L) = \theta(2\mathbf{I} - \Lambda_L)$ so that:

$$f(\mathbf{L}) = \mathbf{Q}f(\Lambda_L)\mathbf{Q}^T \approx \theta(2\mathbf{I} - \mathbf{L}) = \theta(\mathbf{I} + \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}) \tag{4}$$

Since this matrix also has [45] eigenvalues in [0,2], to avoid numerical instability when this matrix is applied multiple times, [20] introduces another “normalization trick” so that instead of $\theta(\mathbf{I} + \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2})$, the expression $\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$ is used instead to have eigenvalues in [− 1,1]:

$$\mathbf{G}_\theta = \tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2} := \hat{\mathbf{A}} \tag{5}$$

Each node v_i in the graph is associated with a c-dimensional feature vector $\mathbf{x}_i \in \mathbb{R}^c$ for each node. The collection of all node features is represented by the feature matrix $\mathbf{X} = [\mathbf{x}_0, \mathbf{x}1, \ldots, \mathbf{x}_{n-1}]^T$, where $\mathbf{X} \in \mathbb{R}^{n \times c}$. Therefore, the aggregation operation involves applying the linear function $\hat{\mathbf{A}}$ to the feature matrix $\mathbf{X}$.

[4] The choice of this type of approximation derives from the fact that it minimizes the maximum error of an approximation to a continuous function on the interval [-1,1]. The order-k Chebyshev polynomial can be defined recursively as $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$ with $T_0(x) = 1$ and $T_1(x) = x$.

[5] involving $\mathbf{A}^k$ or $\mathbf{L}^k$ implies considering nodes k-hop distant, thus possibly creating too much overfitting in the information aggregation.

$$AGGREGATE^{(0)}_{GCN}(\mathbf{A}, \mathbf{X}) := \hat{\mathbf{A}}\mathbf{X} \tag{6}$$

or, for a generic layer l, the (hidden embeddings) feature matrix $\tilde{\mathbf{H}}^{(l)}$ can be obtained from the embeddings of the layer l as (recalling that $\mathbf{H}^0 := \mathbf{X}$):

$$\tilde{\mathbf{H}}^{(l)} := AGGREGATE^{(l)}_{GCN}(\mathbf{A}, \mathbf{H}^{(l)}) := \hat{\mathbf{A}}\mathbf{H}^{(l)} \tag{7}$$

Layer-wise propagation

For a GCN constituted of **L** layers, the "combine" rule in the layer-wise forward propagation is defined as:

$$\mathbf{H}^{(l+1)} := COMBINE^{(l)}_{GCN}(\tilde{\mathbf{H}}^{(l)}) := \sigma_l\left(\tilde{\mathbf{H}}^{(l)}\mathbf{W}^{(l)}\right) = \sigma_l\left(\hat{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) \quad \text{for } l = 0, \ldots, L-1 \tag{8}$$

Where:

- $\hat{\mathbf{A}}$ is the aggregation matrix (defined in the previous subsection)
- $\mathbf{H}^{(l)}$ is the current (hidden) feature vector for layer l.
- $\mathbf{W}^{(l)}$ is the weight learnable matrix.
- σ_l is the activation function (e.g., *ReLU* or *Softmax*).

Initially, $\mathbf{H}^{(0)} = \mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}]^T$. The ReLU function is $ReLU(x) := max(0, x)$. It produces non-linearity by generating zero for negative values and not modifying positive values. In the classical GCN model, the ReLU function is an activation function in the input and hidden layers. The Softmax function is defined (for $\mathbf{x} \in \mathbb{R}^d$) as $Softmax(\mathbf{x}) := exp(\mathbf{x}/T)$ with $T := \sum_{i=0}^{d-1} exp(x_i)$, where d is the dimension of x during the several iterations of the inference. In the classical GCN model, the Softmax is typically used in the output layer for $l = L-1$ (Fig. 2). By definition, Softmax has the property that the sum of all the vector components equals 1, making it a good choice to represent a probability distribution among the predicted classes. The output layer is also indicated as the *READOUT* layer:

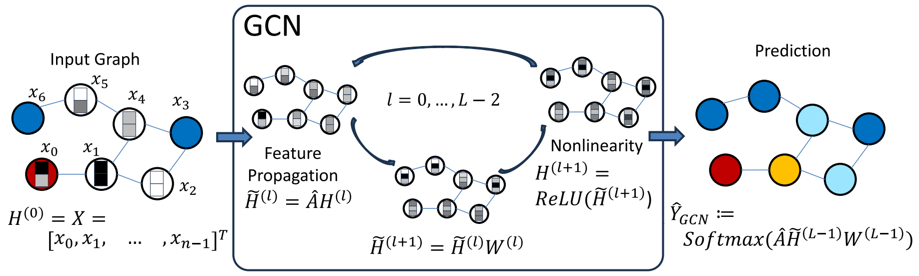
Input Graph
x_6
x_5
x_4
x_3
x_0
x_1
x_2
$H^{(0)} = X = [x_0, x_1, \ldots, x_{n-1}]^T$
GCN
Feature Propagation
$\widetilde{H}^{(l)} = \hat{A}H^{(l)}$
$l = 0, \ldots, L-2$
$\widetilde{H}^{(l+1)} = \widetilde{H}^{(l)}W^{(l)}$
Nonlinearity
$H^{(l+1)} = ReLU(\widetilde{H}^{(l+1)})$
Prediction
$\hat{Y}_{GCN} := Softmax(\hat{A}\widetilde{H}^{(L-1)}W^{(L-1)})$

Fig. 2 GCN propagation workflow: the feature vectors $\mathbf{H}^{(l)}$ are transformed through the $L-2$ layers and by the last (output) layer, which acts as classifier [49]. Here $\hat{A}$ is the aggregation matrix, and W is the weight matrix

$$\hat{\mathbf{Y}}_{GCN} := READOUT_{GCN}(\mathbf{H}^{(L-1)}) := Softmax(\hat{\mathbf{A}}\mathbf{H}^{(L-1)}\mathbf{W}^{(L-1)}) \tag{9}$$

By stacking multiple graph convolutional layers and possibly incorporating pooling layers or skip connections, GCNs can learn hierarchical representations of graph-structured data, enabling them to effectively perform node classification, link prediction, and graph classification tasks.

The goal of a GCN is to learn a function that maps the input node features $\mathbf{X}$ to a set of n predicted output representations (so-called "labels") $\hat{\mathbf{Y}} = [\hat{\mathbf{y}}_0, \hat{\mathbf{y}}_1, ..., \hat{\mathbf{y}}_{n-1}]^T$, where each $\hat{\mathbf{y}}_i \in \mathbb{R}^f$ is a refined representation of the feature vector for each of the f feature-maps (or output classes or output labels), that is $\hat{\mathbf{Y}} \in \mathbb{R}^{n\times f}$ where f is the size of the feature vectors $\hat{\mathbf{y}}_i$ of the output layer $L-1$.

Training a GCN involves optimizing the network's parameters (weights and biases) to minimize a loss function, typically defined based on some related task (e.g., node classification, link prediction, graph prediction). This optimization is often done using backpropagation and gradient descent techniques to update the learnable weight matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{h_l \times h_{l+1}}$ for $l = 0, \ldots, L-1$ where $h_0 = c$ and $h_L = f$. For completeness, a learnable bias vector (typically indicated as $\mathbf{b}^{(l)}$) should be added to $\tilde{\mathbf{H}}^{(l)}\mathbf{W}^{(l)}$. However, for the sake of simplicity of the notation, we omit $\mathbf{b}$ in the following expressions.

The classical GCN model

Overall, GCNs leverage the graph structure to aggregate effectively and combine information across nodes, enabling them to learn powerful representations for tasks on graph-structured data. In the work of Kipf and Welling, the output-layer activation function, i.e., the *READOUT* function, is $sigma_{L-1}() = Softmax()$, while $\sigma_l() = ReLU()$ for the input and hidden layers ($l = 0, \ldots, L-2$). This can be summarized as follows and sketched in Fig. 2:

$$\begin{aligned}&\mathbf{H}^{(0)} = \mathbf{X} \rightarrow \\ &\tilde{\mathbf{H}}^{(l)} = \hat{\mathbf{A}}\mathbf{H}^{(l)} \rightarrow \quad \tilde{\mathbf{H}}^{(l+1)} = \tilde{\mathbf{H}}^{(l)}\mathbf{W}^{(l)} \rightarrow \quad \mathbf{H}^{(l+1)} = ReLU(\tilde{\mathbf{H}}^{(l+1)}) \quad \text{for } l = 0, ..., L-2 \\ &\rightarrow \hat{\mathbf{Y}}_{GCN} := Softmax(\hat{\mathbf{A}}\mathbf{H}^{(L-1)}\mathbf{W}^{(L-1)})\end{aligned} \tag{10}$$

In the proposed semi-supervised classification case [20], two GCN-layers (and one input layer) are used so that Equation 10 for $L = 2$ becomes:

$$\hat{\mathbf{Y}}_{GCN2} := Softmax(\hat{\mathbf{A}}ReLU(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)})\mathbf{W}^{(1)}) \tag{11}$$

The GCN model offers both high accuracy and computational efficiency. Compared to previous models, it avoids the $O(n^3)$ complexity of dense-matrix multiplication, requiring only $O(|\mathcal{E}|fhc)$ multiplications due to the sparsity of $\hat{\mathbf{A}}$ in Equation 11, where $\mathbf{X}$ may still be dense and h denotes the hidden layer size [20].

GCN backpropagation

The backpropagation phase, as illustrated in [37], necessitates a gradient matrix $\mathbf{N}^{(L-1)} \in \mathbb{R}^{n\times f}$, computed as

$$\mathbf{N}^{(L-1)} = \nabla_{\mathbf{H}^{(L-1)}} J \odot \sigma'(\tilde{\mathbf{H}}^{(L-1)})$$

where, given a loss function J, $\nabla_{\mathbf{H}^{(L-1)}} J$ is the matrix of its derivatives with respect to output features in $\mathbf{H}^{(L-1)}$, and $\sigma'(\cdot)$ represents the derivative of the activation function. Gradient matrices for the preceding layers for $l = L-2, \ldots, 0$ are calculated recursively as

$$\mathbf{S}^{(l)} = \hat{\mathbf{A}}\mathbf{N}^{(l)}(\mathbf{W}^{(l)})^T, \qquad \mathbf{N}^{(l-1)} = \mathbf{S}^{(l)} \odot \sigma'(\tilde{\mathbf{H}}^{(l-1)}) \tag{12}$$

In Equation 12, the product $\hat{\mathbf{A}}$ and $\mathbf{N}^{(l)}$ uses Sparse Matrix-Matrix multiplication (SpMM), and the result is multiplied by $(\mathbf{W}^{(l)})^T$ using Dense Matrix Multiplication (DMM). The gradient matrix $\mathbf{N}^{(l)} \in \mathbb{R}^{n \times h_l}$ is utilized to update weight matrix $\mathbf{W}^{(l)}$ through the following formulas:

$$\Delta\mathbf{W}^{(l)} = (\mathbf{H}^{(l-1)})^T \hat{\mathbf{A}}\mathbf{N}^{(l)}, \qquad \mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta\Delta\mathbf{W}^{(l)}$$

Here, $\Delta\mathbf{W}^{(l)}$ is the derivative matrix for the loss function J with respect to the weights in matrix $\mathbf{W}^{(l)}$i, while η is the learning rate. The loss function chosen in [20] is the Cross-Entropy ($CE(\mathbf{Y}, \hat{\mathbf{Y}})$):

$$J = CE(\mathbf{Y}, \hat{\mathbf{Y}}) := -\sum_{i \in \mathcal{V}_0} \sum_{k=0}^{f-1} \mathbf{Y}_{ik} \ln \hat{\mathbf{Y}}_{ik}$$

where $\mathcal{V}_0$ is the subset of the nodes $\mathcal{V}$ that was initially labeled, $\{\mathbf{y}_i\}$ with $i \in \mathcal{V}_0$ are the known label.

GCN variants

After their introduction, intense research has been conducted to increase the prediction accuracy, performance and efficiency of GCNs. For example, on the mathematical side, different functions have been proposed for the aggregation, combine, and readout functions (see Table 3).

Other concepts, like attention mechanisms, e.g., Graph Attention Networks (GAT) [50], sampling, e.g., GraphSAGE [24], simplifications, e.g., SGCN [49] and enhancements, e.g., GraphSAINT [51] have been introduced to improve the accuracy. The many variants have been reviewed in several surveys [26, 36, 42, 44]. However, this survey focuses on implementations that aim to speed up inference and training while working with a limited energy budget, such as in mobile or energy-constrained

Table 3 Examples of variants proposed for the key GCN functions: AGGREGATE, COMBINE, READOUT. $\mathbf{W}_s$:=self-weight matrix, $\mathbf{W}_n$:=neighbor-wight matrix, and $\bar{\mathbf{A}}$ is the sampled $\mathbf{A}$ matrix

Model	Aggregate function $\tilde{\mathbf{H}} = a(\mathbf{A}, \mathbf{H})$	Combine function $\mathbf{H} = c(\tilde{\mathbf{H}}, \mathbf{W})$	Readout function $\hat{\mathbf{Y}} = \sigma_{L-1}(\mathbf{A}, \mathbf{H}, \mathbf{W})$
classic GCN [20]	$\hat{\mathbf{A}} \cdot \mathbf{H}$	$ReLU(\tilde{\mathbf{H}} \cdot \mathbf{W})$	$Softmax(\hat{\mathbf{A}} \cdot \mathbf{H} \cdot \mathbf{W})$
GraphSAGE [24]	$(\mathbf{I} \vert \mathbf{D}^{-1}\bar{\mathbf{A}}) \cdot \mathbf{H}$	$\sigma(\tilde{\mathbf{H}} \cdot (\mathbf{W}_s \vert \mathbf{W}_n))$	Unspecified
GraphSAINT [51]	$(\mathbf{I} \vert \mathbf{D}^{-1}\mathbf{A}) \cdot \mathbf{H}$	$\sigma(\tilde{\mathbf{H}} \cdot (\mathbf{W}_s \vert \mathbf{W}_n))$	Unspecified
FastGCN [52]	$\hat{\mathbf{A}} \cdot \mathbf{H}$	$\sigma(\tilde{\mathbf{H}} \cdot \mathbf{W})$	Unspecified
Cluster-GCN [53]	$(\tilde{\mathbf{D}}^{-1}\tilde{\mathbf{A}} + \lambda \cdot diag(\tilde{\mathbf{D}}^{-1}\tilde{\mathbf{A}})) \cdot \mathbf{H}$	$\sigma(\tilde{\mathbf{H}}\mathbf{W})$	Unspecified
Simplified GCN [49]	$\hat{\mathbf{A}} \cdot \mathbf{H}$	$\tilde{\mathbf{H}} \cdot \mathbf{W}$	$Softmax(\hat{\mathbf{A}} \cdot \mathbf{H} \cdot \mathbf{W})$

systems. Typically, solutions for this category of systems rely on hardware implementations deployed on FPGAs. FPGAs offer the possibility of reusing the hardware for future non-trivial architecture enhancements while exploiting the basic hardware resources for maximum performance. More details about GCNs implementation suitable for FPGAs are presented in the next sections of this paper.

Typical architecture of a GCN-based hardware accelerator

In contrast to traditional deep neural networks, graphs possess highly sparse structures, necessitating fundamentally distinct acceleration strategies.

In Fig. 3, a baseline reference architecture for a GCN accelerator is shown [39]. The systolic array in the combination engine is the main element for efficient dense matrix multiplication. The property buffer provides the current aggregated hidden state ($\tilde{H}^{(l)}$). In contrast, the weight reader provides the learnable matrix $\mathbf{W}^{(l)}$ by feeding the systolic multiplier via the double buffering technique.

The aggregation engine performs a sparse matrix multiplication in which rows of $H^{(l)}$ corresponding to a given vertex are multiplied for the corresponding edge weights in $\hat{\mathbf{A}}_i$ and the result is accumulated using SIMD cores. To exploit the sparsity of the $\hat{\mathbf{A}}$ matrix, e.g., the *Compressed Sparse Row* (CSR) format is used.

The "vertex-prefetch" unit provides the pointers to the "edge-prefetch" and the "feature-reader" units, while the edge-prefetch unit supplies CSR column indices to the feature-reader and the SIMD cores. As the feature-reader collects rows of $H^{(l)}$ for each edge, typical irregular access patterns are observed. Therefore, this phase is memory intensive and quite difficult to optimize.

One approach to alleviate this memory bottleneck is employing a large on-chip Global Cache (gray block in the middle of Fig. 3). Multiple engines share that cache. However, the wide feature matrix often surpasses the cache size, hindering locality exploitation. The Configuration Controller (green block in Fig. 3) collects statistics (e.g., number of feature accesses) from the feature reader and oversees the configuration for the vertex prefetch unit to define the chunk size to be processed.

Several works proposed and evaluated different architectures of a GCN-based accelerator. These works are analyzed in the next sections.

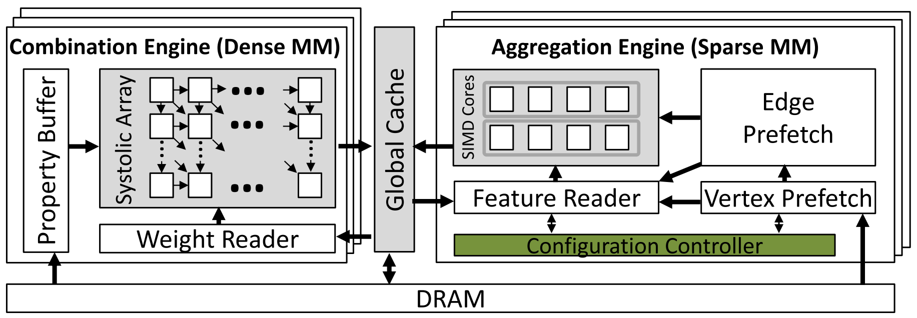

Fig. 3 A baseline reference architecture for a GCN accelerator: the Aggregate and Combine functions are mapped directly to their corresponding hardware engines [39]

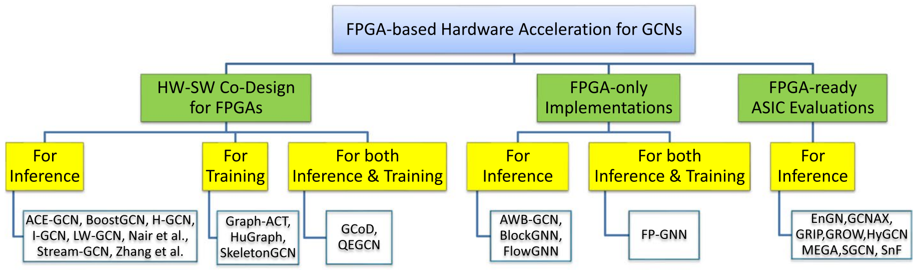
FPGA-based Hardware Acceleration for GCNs
HW-SW Co-Design for FPGAs
FPGA-only Implementations
FPGA-ready ASIC Evaluations
For Inference
For Training
For both Inference & Training
For Inference
For both Inference & Training
For Inference
ACE-GCN, BoostGCN, H-GCN, I-GCN, LW-GCN, Nair et al., Stream-GCN, Zhang et al.
Graph-ACT, HuGraph, SkeletonGCN
GCoD, QEGCN
AWB-GCN, BlockGNN, FlowGNN
FP-GNN
EnGN,GCNAX, GRIP,GROW,HyGCN MEGA,SGCN, SnF

Fig. 4 A taxonomy of the works reviewed in this survey. HW-SW Co-design relates to the techniques that require a combination of optimization both in the host (typically pre-processing) and in the FPGA. FPGA-Only refers to the works proposing a pure FPGA accelerator solution. ASIC Implementations refer to the systems synthesized at the RTL level via silicon compilation tools

Overview of existing solutions

Graph Convolutional Networks (GCNs) have become a highly effective approach for handling graph-structured data across multiple domains, including natural language processing, social network analysis, and image processing [20].

Recent advances in hardware accelerators have shown promising results in accelerating the computation of Graph Convolutional Networks (GCNs) [8, 19, 54, 55]. However, GCNs can be computationally expensive, as they require a large number of matrix multiplications and other operations [18]. Consequently, there has been increasing interest in creating hardware accelerators for GCNs. This computational burden can be mitigated through various acceleration approaches. One such strategy involves hardware-software co-design (see Tables 4, 5), where specific tasks are offloaded to hardware accelerators such as Field-Programmable Gate Arrays (FPGAs). FPGAs, with their fine-grained computational capabilities, the high degree of parallelism and inherent programmability, offer a versatile platform to accelerate both GCN inference and training tasks.

Other approaches focus on executing the GCN computation on FPGAs-only accelerators, or specialized Application-Specific Integrated Circuits (ASICs) to improve performance and energy efficiency in different scenarios, such as real-time processing and resource-constrained edge-computing platforms (see Tables 6, 7, 9, 8).

This section explores studies in the literature that focus on Hardware-Software co-design FPGA-based GCN accelerators, FPGA-only implementations, and ASIC designs, providing insights into their design aspects and performance considerations.

Taxonomy

This survey reviews various approaches to accelerating GCNs using hardware accelerators. It highlights the computational challenges of GCNs due to their reliance on matrix multiplications and discusses how hardware can be leveraged to improve performance and efficiency.

Our proposed taxonomy (Fig. 4) focuses on three main categories of hardware acceleration for GCNs:

Table 4 Main contributions of works in literature that accelerate GCN using Hardware-Software co-design for FPGAs

Work	Main contributions
ACE-GCN [8]	Exploits subgraph similarity and feature exchangeability for faster inference
Boost-GCN [4]	Centralized load balancing engine and phase-level balancing
GCoD [16]	Efficiency aware and resource aware pipelines
GraphACT [19]	Processing pipeline for balance load and efficient utilization
H-GCN [9]	Heterogeneous processing elements for different graph subgraphs
HuGraph [15]	FPGAs in a cluster for parallel GCN training with balanced workloads
I-GCN [5]	Data locality improvement through "islandization" to reduce off-chip memory accesses
LW-GCN [11]	Software preprocessing for data compression and workload balancing
Nair et al. [12]	Dynamically reconfigurable compute core that can alternate between aggregation and transformation.
QEGCN [10]	Quantization and data compression for efficient processing on edge devices
SkeletonGCN [14]	Hardware-software co-design for efficient GCN training on FPGAs
Stream-GCN [56]	Optimizes for small graphs with techniques like pipelining and workload distribution
Zhang et al. [18]	Dense systolic array and non-linear activation module for combination phase

Table 5 Main characteristics of works in literature that accelerate GCN inference and training using Hardware-Software co-design for FPGAs

Work	Year	Inference/training[1]	Data preprocessing[2]	Supported algorithms[3]	Hardware[4]
ACE-GCN [8]	2021	Inference	Jaccard Similarity	GCN	Stratix 10 SX
Boost-GCN [4]	2021	Inference	–	GCN	Stratix 10 GX
GCoD [16]	2022	I+T	Split-and-conquer strategy	GCN, GraphSAGE	AMD Xilinx VCU128
GraphACT [19]	2020	Training	Subgraph mini-batch	GCN, GraphSAGE	AMD Xilinx Alveo U200
H-GCN [9]	2022	Inference	Graph Reordering	GCN, Vanilla-GCN	AMD Xilinx Versal VCK5000
HuGraph [15]	2022	Training	Full process quantization	GCN, GraphSAGE	AMD Xilinx VCU128
I-GCN [5]	2021	Inference	Islandization	GCN,GraphSAGE, FIN	Stratix 10 SX
LW-GCN [11]	2022	Inference	PCOO compression	GCN, GraphSAGE	AMD Xilinx Kintex-7 K325T
Nair et al. [12]	2023	Inference	Undirected graph matrix pruning	GCN, Vanilla-GCN	Stratix 10 MX
SkeletonGCN [14]	2022	Training	CPCOO compression	GCN	AMD Xilinx Alveo U200
Stream-GCN [56]	2022	Inference	Prune zeros	GCN, SimGNN	AMD Xilinx U50, U280
QEGCN [10]	2022	I+T	Quantization and compression	GCN, SAGA-NN	AMD Xilinx VCU128
Zhang et al. [18]	2020	Inference	Graph partitioning, sparsification, reordering	GCN	AMD Xilinx Alveo U200

[1] Inference: Inference task addressed; Training: Training task address I/T: both training and inference tasks addressed

[2] Data Preprocessing technique adopted to optimize data

[3] GCN model implemented

[4] Hardware accelerator used for experiments and evaluation

Table 6 Main contribution of works in literature that accelerate GCN inference and training using FPGA-only designs as hardware accelerators

Work	Main contribution
AWB-GCN [1]	Addresses sparsity with a dedicated matrix multiplication engine for zero-skipping
BlockGNN [6]	Uses block-circulant weight matrices for efficient matrix-vector multiplication
FlowGNN [7]	Generic and reconfigurable dataflow architecture for various GNN models
FP-GNN [2]	Provides an adaptive accelerator framework for both aggregation and combination phases

Table 7 Main characteristics of works in literature that accelerate GCN inference and training using FPGA-only implementations

Work	Year	FPGA/ASIC	Inference/ Training[1]	Data preprocessing[2]	Supported algorithms[3]	Hardware[4]
AWB-GCN [1]	2020	FPGA	Inference	CSC compression	GCN	Stratix 10 SX
BlockGNN [6]	2021	FPGA	Inference	Block-circulant weight matrices	GCN, Graph-SAGE, GAT	AMD Xilinx ZC706
FlowGNN [7]	2023	FPGA	Inference	-	GCN, GAT, PNA	AMD Xilinx Alveo U50
FP-GNN [2]	2022	FPGA	I+T	Graph partition-ing	GCN, GAT, GraphSAGE	AMD Xilinx VCU128

[1] Inference: Inference task addressed; Training: Training task address I/T: both training and inference tasks addressed

[2] Data Preprocessing technique adopted to optimize data

[3] GCN model implemented

[4] Hardware accelerator used for experiments and evaluation

[5] nm: nanometers

Table 8 Main contribution of works in literature that accelerate GCN inference and training using FPGA-ready ASIC evaluations

Work	Main contribution
EnGN [57]	Graph-aware and ring-edge-reducer (RER) dataflow implementation to handle vertices with vary-ing dimensional properties
GCNAX [54]	Cross-dataset analysis for optimized ASIC implementation
GRIP [58]	Leverages a node-flow data structure for ASIC-based inference
HyGCN [3]	Edge-centric aggregation that divides vertex workload into sub-workloads. Matrix-Vector Mul-tiplication (MVM) and the dense systolic array are used in the combination phase. Grouping of Single Instruction Multiple Data (SIMD) cores and Processing Elements (PE)
SGCN [13]	Utilizes BEICSR compression for efficient inference on an ASIC.
SnF [39]	Dynamic tiling and automatic tile morphing
GROW [59]	HDN cache for graph nodes with high degree and a "row-stationary" dataflow leveraging Gustav-son's algorithm
MEGA [60]	Degree-Aware mixed-precision quantization with Adaptive-Package format and Condense-Edge scheduling for optimized storage and data locality

- *Hardware-Software Co-Design (HW-SW Co-Design) for FPGAs*. This approach combines software and hardware techniques to optimize GCN execution. It often involves preprocessing the graph data on the software side and designing specialized hardware architectures on FPGAs to handle specific GCN operations.

Table 9 Main characteristics of works in literature that accelerate GCN inference and training using FPGA-ready ASIC evaluations

Work	Year	FPGA/ASIC	Inference/ training[1]	Data preprocessing[2]	Supported algorithms[3]	Hardware[4]
EnGN [57]	2021	ASIC	Inference	–	GCN	RTL-Simulation
HyGCN [3]	2020	ASIC	Inference	–	GCN	Synopsys at 12nm[5]
GCNAX [54]	2021	ASIC	Inference	cross-dataset analysis	GCN	Synopsys at 40nm[5]
GRIP [58]	2023	ASIC	Inference	node-flow data structure	GCN	SystemVerilog 28nm[5] CMOS
SnF [39]	2022	ASIC	Inference	–	GCN	Verilog/Cadence at 45nm[5]
SGCN [13]	2023	ASIC	Inference	BEICSR compression	GCN, GraphSAGE	Synopsys at 45nm[5]
GROW [59]	2023	ASIC	Inference	–	GCN	Synopsys at 65nm
MEGA [60]	2024	ASIC	Inference	Adaptive-Package storage format	GCN, GIN, GraphSAGE	Synopsys at 28nm

[1] Inference: Inference task addressed; Training: Training task address I/T: both training and inference tasks addressed

[2] Data Preprocessing technique adopted to optimize data

[3] GCN model implemented

[4] Hardware accelerator used for experiments and evaluation

[5] nm: nanometers

- *FPGA-only Implementations.* This category utilizes FPGAs as hardware accelerators without relying on software co-design. FPGAs offer fine-grained parallelism and programmability, making them suitable for accelerating GCN operations like sparse matrix multiplications.
- *FPGA-ready ASIC Evaluations.* This category includes the FPGA-synthesizable designs that have been evaluated with ASIC[6] design tools such as Synopsis. ASICs evaluations could show higher performance and lower power consumption for GCN inference.

This section provides detailed explanations and comparisons of several research works within each category. Some key takeaways from the highlighted works are in Tables 4, 6, and 8.

Overall, hardware acceleration offers significant performance and efficiency improvements for GCNs. The choice of approach depends on factors like cost, flexibility, power consumption, and the specific GCN application.

Hardware-software co-sesign for FPGAs

This subsection illustrates the works in literature that adopt Hardware-Software co-design to improve the performance of GCN training and inference execution. The main characteristics of the works presented in this section are summarized in Tables 4 and 5.

ACE-GCN [8] investigates the characteristics of real-world graphs, which exhibit pronounced power-law distributions and significant sparsity to enhance graph processing

[6] Application-Specific Integrated Circuits (ASICs) are custom-designed chips optimized for a specific task, however production costs is only convenient for very large number of devices and they less flexible than FPGA.

speeds. The main objectives are to decrease the neural network's computational load and speed up inference by leveraging subgraph similarity and feature interchangeability. The similarity core is based on a vectorized Jaccard graph similarity coefficient, enabling the accelerator to identify and exploit similar graph structures. This optimization enables ACE-GCN to tailor its operation according to resource availability and targeted specifications.

Boost-GCN [4] is a framework that aims to optimize GCN inference on FPGA. Boost-GCN introduces three primary contributions: first, the "Hardware-aware Partition-Centric Feature Aggregation" (PCFA) scheme, which utilizes a "3D partitioning" aligned with the vertex-centric computing approach, enhancing data reuse and reducing external memory communication volume. It also address imbalanced workload with a custom load-balancing engine. Second, a new hardware design enables pipelined execution across two distinct computation phases in order to minimize pipeline stalls. Lastly, BoostGCN offers a full FPGA-based GCN acceleration framework, equipped with optimized RTL templates that facilitate the generation of hardware designs customized for different GCN models. Furthermore, BoostGCN employs two types of Feature Update Modules (FUMs): Sparse-FUM for low-density feature matrices (less than 10%) and Dense-FUM for high-density matrices. Sparse-FUM processes batches of feature vectors similar to FAM, while Dense-FUM utilizes a 2D systolic array for batch multiplication with weight matrices. The choice between Sparse-FUM and Dense-FUM depends on the density of the input feature matrix, determined by a threshold (Tth). Sparse-FUM is used when the density is below Tth, while Dense-FUM is applied for higher densities.

GCoD [16] or "Graph Convolutional Network Acceleration" through "Dedicated Algorithm and Accelerator Co-Design", enhances GCN inference by addressing the challenges posed by irregular and sparse graph datasets. This is accomplished using a split-and-conquer training approach, which divides the graphs into either denser or sparser local neighborhoods. This method generates adjacency matrices with balanced workloads and enhances communication between accelerators. In addition, the FPGA implements both efficiency-aware and resource-aware pipelines, offering advantages such as increased data reuse through direct reuse of intermediate results and reduced on-chip storage needs by storing only one column of aggregation outputs. GCoD ultimately increases efficiency, minimizes data access, and reduces processing workloads, resulting in improved GCN performance during training up to 7.8x and 2.5x speedup compared to HyGCN and AWB-GCN, respectively.

GraphACT [19] tries to improve GCN training on heterogeneous CPU+FPGA platforms by strategically assigning computational tasks and managing data storage. The CPU is responsible for graph sampling and loss gradient calculations, while the FPGA handles both forward and backward propagation along with the operations of combination and aggregation. GraphACT adopts a subgraph-based mini-batch algorithm to minimize CPU-FPGA communication, identifying and removing repetitive aggregation operations on shared node neighbors. GraphACT analyzes feature propagation by leveraging graph theory to identify frequently occurring node sets. These sets' vector sums are pre-computed on the CPU to reduce on-chip operations and FPGA memory accesses. Additionally, GraphACT parallelizes key training steps with optimized on-chip computation modules, integrating them into a processing pipeline for balanced load and efficient utilization of FPGA resources.

H-GCN [9] enhances GCN inference performance by partitioning the graph into three distinct subgroups: densely connected clusters, sparsely connected regions, and isoleted nodes. Each of these subgraphs is processed by specialized hardware of a heterogeneous accelerator platform (e.g., ”AMD Xilinx Versal Adaptive Compute Acceleration Platform“). Specifically, densely connected components are handled by dense Artificial Intelligence Engines (AIEs), while sparse AIEs manage components with looser connections, and scattered nodes are addressed using programmable logic (PL). H-GCN capitalizes on the sparsity support of AIEs through a density-aware method, efficiently mapping sparse matrix-matrix multiplication (SpMM) tiles onto the systolic tensor array. Furthermore, H-GCN employs a combination of acceleration techniques, including PL, AIEs, and systolic tensor arrays, to accelerate dense and sparse matrix operations within GNNs. This heterogeneous approach enables H-GCN to process diverse subgraph types efficiently, thus achieving high-performance GCN inference. H-GCN shows significant performance improvements compared to leading GCN accelerators and general-purpose processors. Compared to I-GCN, H-GCN achieves up to 2.3× speedup. Additionally, H-GCN exhibits 1.12× and 1.64× higher energy efficiency than I-GCN and AWB-GCN, respectively.

HuGraph [15] is a framework designed to implement GCN training across a cluster of heterogeneous FPGAs. In this system, FPGAs utilize a 1D ring topology with synchronous data parallelism for efficient operation. HuGraph aims to improve the performance of GCN training with three main strategies. Firstly, HuGraph implements a quantization process for data-parallel training using neighbor sampling, aimed at lowering both computational and memory demands. Secondly, it introduces an custom balanced sampling technique to distribute tasks evenly across heterogeneous FPGAs, ensuring that less resourceful FPGAs do not hinder the overall performance of the cluster. Thirdly, the execution sequence of GCN training is organized by HuGraph to reduce time delays, focusing on executing the most impactful operations first. This is done by prioritizing the execution of operations with the greatest impact on performance. Experimental results show that HuGraph achieves speedups of up to 102.3x, 4.62x, and 11.1x compared to the most advanced CPU, GPU, and FPGA platforms, respectively, with only a slight reduction in accuracy.

I-GCN methodology, as delineated in [5], introduces an approach to enhance the efficacy of GCNs by increasing data locality through a technique called “islandization”. This method partitions a graph into smaller clusters, referred to as islands, interconnected via hub nodes. Using this configuration, I-GCN improves computation performance by increasing data locality and reducing off-chip access. This procedure is executed directly on the hardware, eliminating the necessity for prior graph data preprocessing. Additional details regarding the I-GCN architecture can be found in Section Proposed Solutions: Architecture Highlights of Selected FPGA-based GCNs.

LW-GCN [11] is an FPGA-based GCN accelerator specifically designed for high energy efficiency and low latency. It utilizes a co-optimization approach between software and hardware to address challenges in mapping GCN algorithms onto hardware. A preprocessing software algorithm transforms the sparse matrix into a “Packet-level

Column-only coordinate list" (PCOO) format, thereby decreasing the needs for storage and bandwidth. In addition, the algorithm optimizes GCN processing by balancing the workload on different processing elements (PE), reducing computation imbalance. At the hardware level, LW-GCN employs a multi-bank dense data memory architecture that incorporates data replication to minimize data collision incidents. Additionally, its microarchitecture utilizes a round-robin method to allocate non-zero elements to PEs, ensuring an even distribution of computational tasks. LW-GCN concatenates multiple rows before assigning them to a PE to manage scenarios where the count of non-zero elements within a row differs significantly from that of other rows.

Nair et al. [12] proposed a hardware-software co-design approach to address the challenges of accelerating GCNs with undirected graphs. Within the software layer, a preprocessing stage reorganizes the edges and features to align with the custom dataflow architecture, improving the consistency of memory access and data reuse during the aggregation phase. Custom dataflow and reconfigurable compute cores are designed to exploit the inherent symmetry of the adjacency matrix in undirected graphs. In undirected graphs, the upper triangle of the adjacency matrix mirrors the lower triangle's transpose. Consequently, both triangles do not need to be computed separately. This method improves data transfers and reduces data usage. Additionally, Nair et al. implement a computing core that can alternate between aggregation and transformation, depending on the computational requirements, minimizing imbalances in GCN tasks.

QEGCN [10] is a hardware accelerator architecture suited for GCN inference using edge-level parallelism on FPGA platforms. Indeed, QEGCN is the first GCN accelerator to employ quantization-aware training, which involves converting the network parameters and input feature vectors into lower precision formats. This approach reduces the model size and enhances inference speed while maintaining accuracy. Quantization is performed in the preprocessing phase, along with compression of the feature vector matrix and edge block partitioning, which enables independent handling of each block by distinct processing elements. Furthermore, the architecture boosts GCN performance at the edge by optimizing the algorithm, integrating a pipeline framework specifically designed for edge computing, and employing atomic operations to ensure effective load distribution. In this work, the FPGA performs several crucial functions. It serves as the primary tool for processing data for the QEGCN model. It significantly accelerates the inference process by leveraging its parallel processing capabilities, thereby increasing the speed of predictions made by the QEGCN model. Further details on the QEGCN architecture can be found in Section Proposed Solutions: Architecture Highlights of Selected FPGA-based GCNs.

SkeletonGCN as detailed in [14], is an FPGA-based accelerator designed for efficient GCN training, integrating multiple strategies to optimize GCN efficiency and performance on FPGA platforms. For example, a strategy quantizes the GCN features and adjacency matrices to SINT16, minimizing storage needs and computational overhead. Moreover, a linear-time algorithm for compressing sparse matrices is utilized to reduce memory bandwidth demands while facilitating efficient hardware decompression. Sparse adjacency matrices are further compressed via the Compact PCOO format (CPCOO), which reduces memory consumption compared to the conventional Packed COO (PCOO) format. The CPCOO format segments sparse matrices into header and body sections, tackling inefficient memory utilization of empty rows in extensive datasets. In comparison to PCOO, CPCOO achieves a reduction in memory usage ranging from 2.87× to 4.81× across different datasets. Finally, SkeletonGCN adopts a unified hardware architecture capable of handling multiple types of matrix multiplications on the same group of processing elements (PEs), thus maximizing DSP utilization on FPGA.

Stream-GCN [56] is introduced as a versatile architecture tailored to accelerate GCNs, focusing particularly on streaming small graphs. It integrates optimizations specifically tailored for small graphs, such as interlayer pipelining, in situ sparsity utilization, and efficient workload distribution to enhance performance and area efficiency. Using the flexibility of FPGAs, StreamGCN achieves efficient and high-speed processing by implementing deep pipelines with multiple nested levels of parallelization. StreamGCN integrates specific hardware optimizations to minimize global memory accesses. It utilizes scratchpad memory to store matrices that require random local access, reducing the frequency of accessing global memory. Dedicated processing units are tailored for each step of the GCN algorithm, optimizing hardware and reducing global memory operations. An efficient workload distribution mechanism is used to mitigate load-imbalance issues, ensuring uniform distribution across processing units and reducing global memory accesses for data fetching.

Zhang et al. [18] introduces a combined software and hardware approach to accelerate GCNs. Their co-optimization approach for large-scale GCN inference on FPGA involves data partitioning, two-phase preprocessing, and generic FPGA architecture for pipelining aggregation and transformation kernel. The two-phase preprocessing algorithm includes graph sparsification to reduce edge connections of high-degree nodes and node reordering to group adjacent nodes, further improving data locality and significantly reducing external memory accesses. The processed graph is then passed to a FPGA based hardware accelerator, which performs data aggregation and combination with custom and optimized pipelined components. The aggregation module utilizes a sparse array structure, while the combination module employs a dense systolic array along with a non-linear activation component. To minimize latency, the aggregation module implements a technique that allows for simultaneous data processing. Furthermore, the accelerator provides two operational modes that depend on the order of matrix multiplication, each requiring different pipelining approaches to interconnect the various components.

FPGA-only implementations

This subsection discusses various studies that utilize Field Programmable Gate Arrays (FPGAs) as hardware accelerators to enhance the performance of GCNs. A summary of the key characteristics of these studies can be found in Tables 6 and 7.

AWB-GCN [1] accelerator tackles the inherent sparsity in GCNs through the creation of a specialized matrix multiplication engine designed for effective zero-skipping. It utilizes a Task Distributor and Queue (TDQ) system to channel data from memory to processing elements (PEs) and accumulators, offering two configurations optimized for different levels of sparsity. AWB-GCN focuses on processing combinations before aggregation to minimize the number of operations. Through fine-grained pipelining, AWB-GCN concurrently handles both the combination and aggregation operations within a single layer. Three workload balancing functions distribute workload among PEs, handle minor imbalances through dynamic redistribution, and address major imbalances by remapping non-zero elements. Hardware autotuning techniques, including load distribution adjustment, remote toggling, and row reorganization, ensure a balanced workload across PEs, maximizing utilization and minimizing synchronization overhead. Moreover, AWB-GCN exploits parallelism, data reuse, and reduced memory access latency by adopting pipelined Sparse Matrix-Matrix Multiplications (SpMMs) and interlayer data forwarding with matrix blocking. AWB-GCN achieves a significant performance speedup over CPUs, GPUs, and prior GCN accelerators, primarily due to its hardware-based autotuning framework. A detailed description on the AWB-GCN implementation can be found in Section Proposed Solutions: Architecture Highlights of Selected FPGA-based GCNs.

BlockGNN [6] tries to improve the performance of real-time inference on resource-constrained edge-computing platforms. BlockGNN introduces block-circulant weight matrices as a compression technique to reduce the computational complexity of graph neural networks (GNNs), including GCNs. This compression technique facilitates matrix-vector multiplications using the "Fast Fourier Transform" (FFT), which reduces latency and improves energy efficiency. In fact, block-circulant weight matrices reduce computational complexity from $O(n)$ to $O(nlogn)$ without compromising accuracy. Furthermore, block-circulant matrices are computed using a pipelined "CirCore" architecture in FPGA to improve performance and efficiency further. BlockGNN shows improvements up to 8.3x compared to HyGCN.

FP-GNN [2] introduces an adaptive FPGA accelerator, focusing on graph neural networks (GNNs) acceleration, with a specific emphasis on the aggregate and combination phases. This innovation responds to the growing demand for flexible and efficient GNN acceleration. Central to FP-GNN's design is the Adaptive GNN Accelerator (AGA) framework, which concurrently supports both Aggregation and Combination phases, enhancing flexibility and efficiency in GNN acceleration. AGA incorporates a key concept known as "Adaptive Graph Partition" (AGP). AGP effectively separates the relationship between the chip's memory size and the graph's partition size, enhancing the use of off-chip memory bandwidth. By employing hardware-aware edge leveling and interval interleaving for task scheduling, AGP preprocesses the data to optimize memory access and reduce the overhead of graph repartitioning between GNN layers. When compared to leading solutions, FP-GNN demonstrates average performance efficiency gains of 25.9×, 1.64X, 1.59×, 1.18×, and 0.98× over HyGCN, BoostGCN, AWB-GCN, GCNAX, and I-GCN, respectively. Further information about the FP-GNN architecture is provided in Section Proposed Solutions: Architecture Highlights of Selected FPGA-based GCNs.

FlowGNN [7] introduces a highly adaptable dataflow architecture designed to accelerate graph neural networks (GNN) on FPGAs. Unlike conventional accelerators, FlowGNN's design is generic and reconfigurable, accommodating a wide array of GNN models without extensive architectural modifications (e.g., GCNs). FlowGNN accomplishes this by harnessing an efficient dataflow architecture that maximizes parallelism and reduces idle time within processing units. Furthermore, its multilevel parallelism strategy, message-passing mechanism, and edge-embedding guarantees optimal utilization of hardware resources, thereby enhancing overall performance. Moreover, FlowGNN prioritizes real-time processing, eliminating the need for preprocessing and enabling seamless on-the-fly computation of streamed graph data. FlowGNN surpasses existing GNN accelerators such as I-GCN with a 1.26× speedup and 1.55× energy efficiency across different datasets. More details about the FlowGNN architecture are described in Section Proposed Solutions: Architecture Highlights of Selected FPGA-based GCNs.

FPGA-ready ASIC evaluations

This subsection discusses various studies that attempt to further improve GCN processing performance and energy consumption by designing ASIC-based solutions. A summary of the key characteristics of these studies can be found in Tables 8 and 9.

EnGN [57] stands out among previous graph processors and neural network accelerators due to their unified architecture, capable of efficiently supporting diverse graph neural networks (GNN) models such as GCNs. This versatility is achieved by abstracting typical GNN algorithms into three stages: vertex feature extraction, feature aggregation, and graph update. Its graph-aware and ring-edge-reducer (RER) dataflow implementations efficiently handle vertices with varying dimensional properties and ensure high throughput GNN operations. Moreover, graph tiling and scheduling techniques optimize data locality and minimize memory accesses, enhancing overall performance and energy efficiency. The EnGN unified architecture makes it suitable for processing large-scale GNNs. Indeed, it optimizes memory space management with a specialized on-chip memory hierarchy tailored to the non-uniform distribution of graph vertices. The reconfigurable architecture of EnGN and the interconnects adapt dynamically to varying workload dimensions, maximizing hardware and memory bandwidth utilization. Leveraging FPGA technology, EnGN performs significantly better than similar GCN accelerators like HyGCN [3] (see Fig. 10).

GCNAX Li et al. identify various inefficiencies in existing accelerators, including load imbalance, execution order issues, and loop optimization inefficiencies. To address these challenges, GCNAX [54] is proposed as a flexible accelerator with reconfigurable dataflow, allowing adjustments in loop order and fusion strategy. Additionally, GCNAX optimizes tile sizes based on cross-dataset analysis to enhance resource utilization and minimize data movement, thus improving performance. Through design space exploration, GCNAX adapts its dataflow to match the characteristics of each dataset during inference. Furthermore, GCNAX employs the outer product method to address the imbalanced presence of zeros, a common issue in graph-related computations. This method is particularly suitable for Sparse Matrix Multiplication (SpMM), which effectively reduces unnecessary computations and mitigates workload imbalance. As a result of these techniques, GCNAX demonstrated significant performance improvements compared to other accelerators. It achieves approximately a 6x speedup over HyGCN [3] (see Fig. 10) and a 1.6x improvement in efficiency compared to AWB-GCN [1].

GRIP [58] accelerator is based on the gather/reduce/ transform/activate (GReTA) [61] design to create a versatile accelerator capable of handling GCNs. The GRIP architecture is designed with separate and customizable units and accumulators for edges, including gather and reduce, as well as for vertices, which involve transform and activate. This design enables user-defined functions to handle updates for both edges and nodes. A control unit manages data movements between these units and their respective buffers.

Moreover, GRIP addresses the challenge of high latency penalty GCNs into a node flow data structure, enabling high locality access to graph data and reducing the latency penalty associated with random accesses. Additionally, GRIP achieves low-latency inference through architectural features such as vertex-tiling, which improves weight locality in the combined phase, and specialized execution units for each phase of GCN inference. The GRIP approach significantly reduces latency compared to CPU and GPU implementations while operating with minimal power consumption (almost 5W).

HyGCN [3] introduces a combined architecture for GCNs, consisting of dedicated components for the aggregation and combination stages that are regulated by a control mechanism. The aggregation stage features a sample, edge scheduler, and sparsity eliminator to handle sparsity efficiency. The combination stage employs a dense systolic array approach. HyGCN utilizes edge and "Matrix-Vector Multiplication" (MVM) approaches to take advantage of the GCNs parallelism. The aggregation phase of this model is based on the edge-centric approach, while the MVMs is used for the combination step. This helps the hardware design to support parallel processing. On the other hand, MVM operations in the combination phase are executed in parallel using a modified systolic array design, capitalizing on data reuse and computational parallelism. Multiple systolic modules are integrated for different granular sizes. Furthermore, the architecture adapts to varying workloads by allowing flexible grouping of single instruction multiple data (SIMD) cores and processing elements (PEs).

SGCN [13] aims to improve the GCN execution performance by focusing on the preprocessing phase. Indeed, SGCN compresses the data using the BEICSR format (Bitmap index embedded in place CSR), which decreases the sparsity in intermediate feature matrices of deep GCNs. The BEICSR format is based on bitmaps, in which data and their indices are placed in the same row to improve data locality and reduce off-chip memory traffic. Additionally, BEICSR features in-place compression technology, enabling parallel writing and random access capabilities during GCN operations. The Sparse Aggregation Unit performs the aggregation phase in GCNs using sparse features. It contains SIMD MAC cores that process the accumulation of features from multiple vertices and a parallel prefix sum unit that converts bitmap indices to reversed indices with non-zero values. The combination phase is performed in conjunction with a compressor unit. The compressor writes bitmap indices non-zero values to the memory in an in-place manner, eliminating the need for extra memory traffic.

SnF [39], optimizes the computational patterns found within GCNs by leveraging feature-slicing dataflow as a primary loop. This approach creates a predictable and repetitive access pattern to process random graphs. Furthermore, the feature-slice approach increases the number of slices instead of enlarging the vertex tiles. This improves the operational efficiency of SnF. Moreover, through a mechanism named "automatic tile morphing", SnF dynamically adjusts its tile size based on this repetitive pattern. Near-optimal configurations can be achieved in a few iterations, eliminating the need for extensive offline analysis. Compared with the state-of-the-art, SnF achieves speedups of up to 1.7x and 1.4x compared to HyGCN and AWB-GCN, respectively.

GROW [59] proposes a novel GCN accelerator designed to efficiently handle the memory-intensive operations involved in graph convolutional neural networks. GROW employs a dataflow architecture that maintains "row-stationarity", utilizing principles derived from Gustavson's algorithm [62]. Furthermore, GROW employs an HDN ("High-Degree Node") cache to capture high-locality dense matrix rows to store the IDs of the top-N high-degree nodes. This approach minimizes memory traffic and boosts performance by efficiently managing accesses to the sparse data. This approach minimizes memory traffic, significantly improving performance and energy efficiency compared to existing accelerators. GROW further enhances locality and parallelism through graph partitioning and run-ahead execution, leading to substantial speedups, particularly for large-scale graph datasets.

MEGA [60] proposes a new approach for speeding up and reducing the energy consumption of graph neural networks (GNNs). The paper describes a new accelerator called MEGA, which employs a novel method called "Degree-Aware" mixed-precision quantization to compress the GNN model and reduce memory requirements. The accelerator is designed to exploit the sparsity of GNNs, which is a common challenge for existing GNN accelerators. By implementing the "Adaptive-Package" storage format and a "Condense-Edge" scheduling strategy, the accelerator reduces the amount of data that needs to be transferred to and from DRAM, thus improving performance and energy efficiency. MEGA outperforms cutting-edge GNN accelerators such as HyGCN, GCNAX, GROW, and SGCN with an average speedup of 38.3×, 7.1×, 4.0×, 3.6×, respectively.

Proposed solutions: architecture highlights of selected FPGA-based GCNs

Several solutions propose novel hardware architectures to address challenges such as poor data locality and redundant computation in GCN inference on FPGAs, as discussed in Section Overview of existing solutions. We have selected some advanced representing works including I-GCN [5], AWB-GCN [1], FP-GCN [2], FlowGNN [7] and QEGCN [10] based on their reported performance and popularity among studies of implementing GCN on FPGA hardware.

FlowGNN addresses the GCN implementation challenges by offering a generic and flexible architecture that supports a wide range of GNN models without requiring graph preprocessing or graph-specific optimizations. It features a dataflow architecture that overlaps node transformation and message passing stages, together with different level of parallelism such as node, edge, scatter, and apply. This architecture allows for efficient processing of small graphs with relatively low latency. Figure 5 shows the conceptual workflow of the FlowGNN's architecture.

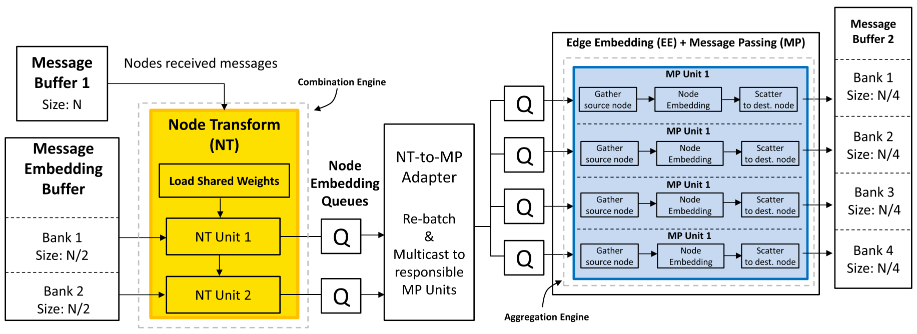

Fig. 5 The dataflow architecture of FlowGNN. [7]

This architecture consists of two key processing parts : the Node Transformation (NT) unit (shown in yellow color, referred to as "combine engine" in Section Problem definition: background and mathematical foundations) and the Message Passing (MP) unit (shown in blue color, referred to as "aggregate engine" in Section Problem definition: background and mathematical foundations). These units are responsible for processing nodes and edges, respectively in the graph. The workflow includes data buffers for node embeddings and messages, allowing for efficient graph data processing. Additionally, a node queue facilitates pipelined NT and MP operations, enabling parallel processing and reducing idle cycles.

FlowGNN introduces significant improvements over the baseline architecture by incorporating multiple levels of parallelism, as illustrated in Fig. 5. This includes parallel NT and MP units, each handling a subset of nodes or edges, and an NT-to-MP adapter facilitating on-the-fly multicasting. The architecture also partitions node embeddings and message buffers into multiple banks to enable parallel access (see Fig. 5). These enhancements allow concurrent processing of nodes and edges, leading to improved performance and efficiency.

The NT unit is responsible for managing node-level calculations essential for GNNs, including the application of fully connected layers and activation functions. It employs embedding-level parallelism to process node embeddings efficiently. The MP unit, on the other hand, manages edge-level computations, including message aggregation and scattering. FlowGNN supports both NT-to-MP and MP-to-NT dataflow configurations, allowing flexibility in processing orders based on the requirements of different GNN models.

I-GCN At the core of I-GCN, there is an innovative online graph restructuring algorithm known as islandization. An island refers to a cluster of nodes within a graph that are densely connected to one another, while having fewer connections to nodes outside

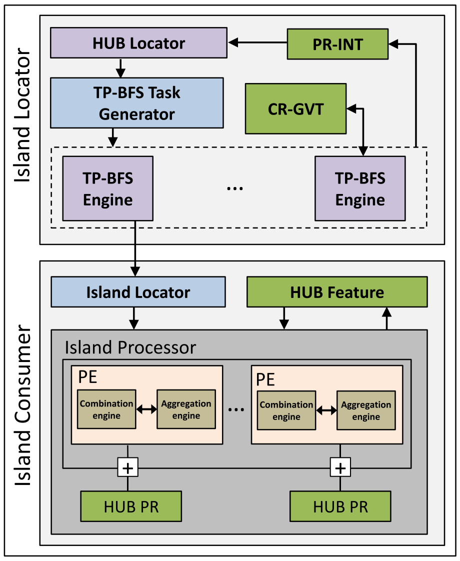

Fig. 6 The overview of the I-GCN. The HUB Locator (HL) identifies hubs, forwarded to TP-BFS Task Generator (TP-BFS-TG). TP-BFS conducts Threshold-based and Parallel BFS. PR-INT denotes to Island Node Table

the cluster and hubs are nodes with high fan (in or out degrees), which act as the points of contact for islands. This algorithm systematically identifies hubs and islands within the graph, leveraging their inherent structural properties to enhance data locality and streamline computation. By seamlessly integrating islandization into its architecture, I-GCN offers a transformative approach to accelerating GCNs, promising significant advancements in performance and efficiency for real-world applications. Fig. 6 shows the overview of I-GCN architecture design.

In Fig. 6, an *Island Locator* hardware module searches for the islands (as well as the hubs and nodes) simultaneously. The Island Locator scans the neighbor nodes of each hub it finds, using varying thresholds to identify hubs and subsequently shaping the island.

Concurrently, the adjacency information of each newly constructed island is sent to the "*Island Consumer*" module. Afterward, "Island Consumer" treats the island as a compact yet dense sub-graph, retrieves its node features, and carries out the necessary aggregation and combination. The "Island Consumer" are capable of processing an island immediately upon its formation, without needing to synchronize at the conclusion of each round; they do not have to delay processing until every island in the islandization round has been established.

FP-GNN Introduces a strategy called "Adaptive Graph Partition" (AGP), which combines several techniques to speedup the Graph processing such as "Hardware-aware Edge Leveling", "Interval Interleaving" for task scheduling, and source node caching. This approach aims to reduce memory bottlenecks and remove the need for graph repartitioning overhead across GNN layers, while fostering both feature-level and node-level parallelism throughout the "Aggregation" and "Combination" stages.

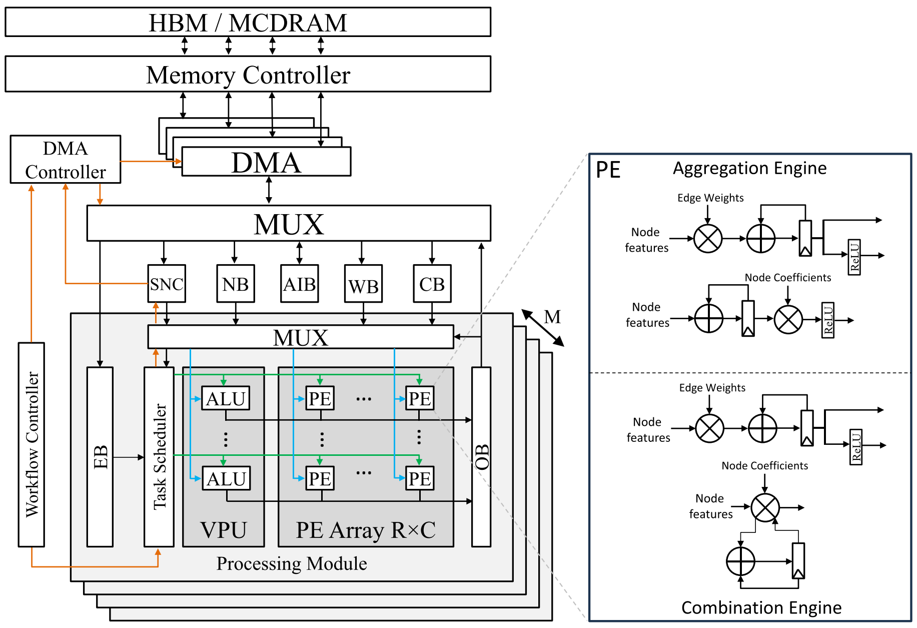

Fig. 7 The architecture overview of the FP-GCN

Fig. 7 depicts the overall architecture design of the FP-GNN. The memory system includes "High-Bandwidth Memory" (HBM) and "Multi-Channel DRAM" (MCDRAM) to ensure fast data access for the accelerator. Moreover, On-chip buffers like Edge Buffer (EB), Source Node Cache (SNC), Node Buffer (NB), Address Index Buffer (AIB), Weight Buffer (WB), Coefficient Buffer (CB), and Output Buffer (OB) store different kinds of data temporarily. The arrows show the different flows in which orange ones are control flow and the other ones are data flow.

These buffers help with various tasks, such as storing edge information, preventing duplicate data requests, storing node features, managing data addresses, storing weights and coefficients, and holding intermediate or final results.

Data movement between the on-chip buffers and off-chip memory is regulated by the Memory Controller in conjunction with the DMA Controller, which handles requests for Direct Memory Access (DMA).

AWB-GCN introduces an innovative and efficient architecture aimed at enhancing the performance of GCNs and Sparse Matrix Multiplication (SpMM) kernels for matrices exhibiting a power-law distribution. It also features a hardware-based framework for autotuning workload distribution, specifically designed to manage significant workload imbalances.

Figure 8 illustrates the architecture of AWB-GCN. Below, we provide a brief overview of its components. The figure depicts a single "Super-PE" alongside four "Labor-PEs". The "Labor-PE" is an adaptation of the normal PE, distinguished by its connection to an adder tree for aggregating results. A caching mechanism aggregates the result of "evil

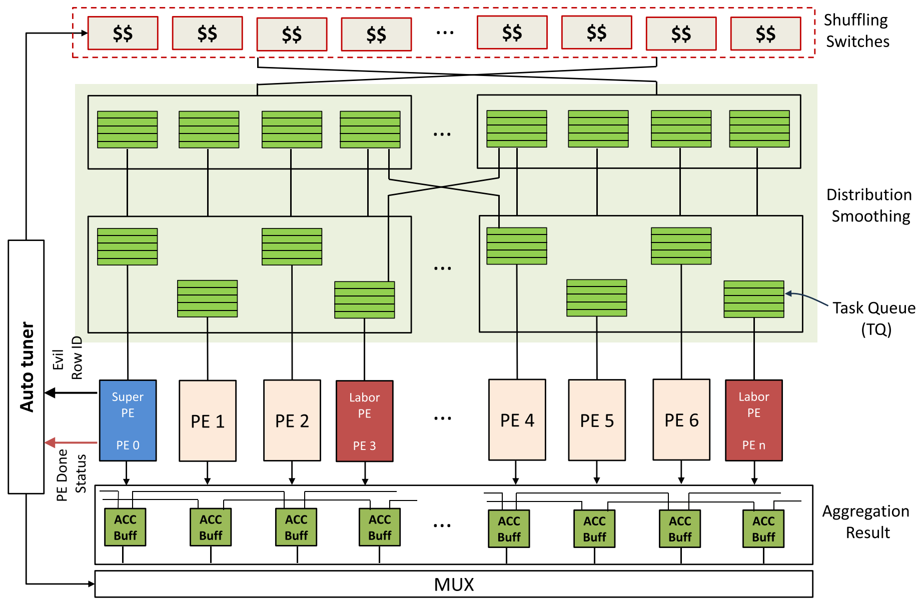

Fig. 8 Overall architecture design of AWB-GCN describing the SpMM engine and other AWB-GCN techniques such as remote switching, evil row remapping and smoothing. In blue color, the Super-PE (Master-PE) is highlighted. In red color, one of the Labor-PE is highlighted

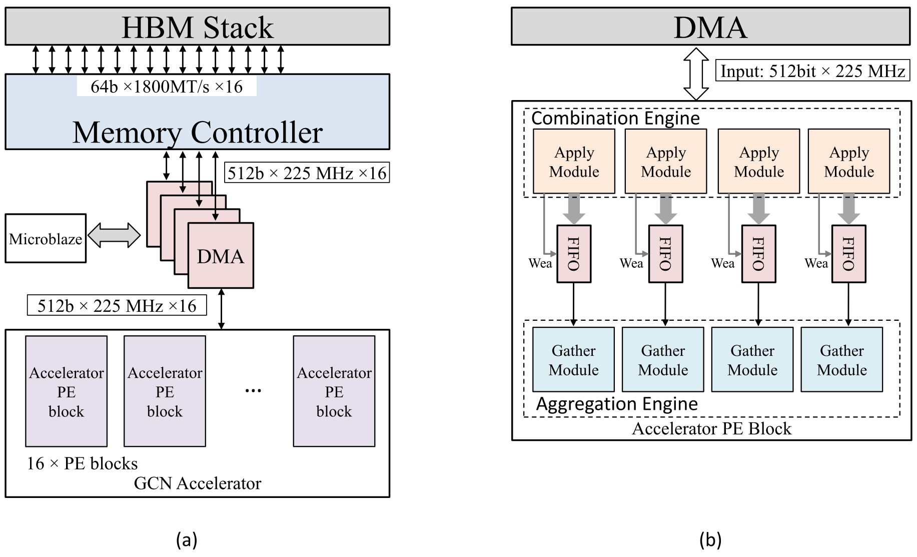

Fig. 9 The overall overview of the QEGCN architecture (**a**) and the detail of Accelerator PE block in (**b**)

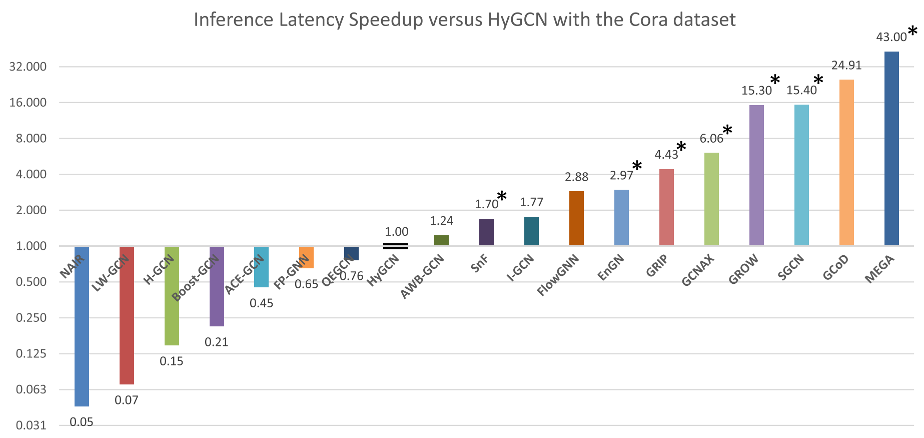

Fig. 10 Inference latency speedup comparison among GCN architectures when using the Cora dataset, considering HyGCN as baseline. FPGA-ready designs that are evaluated by using ASIC tools are marked with an asterisk

rows". On the other hand, the "Super-PE" is significantly larger than the other PEs, as it is tasked with profiling the evil rows.

The "Super-PE" includes two additional modules: a non-zero counter (consisting of a local buffer) that counts the number of non-zeroes in each row and a parallel sorting circuit that keeps track of the rows with the greatest number of non-zeroes. Specifically, the "Super-PE" includes additional modules to effectively sort and count specific kinds of data items. The allocation of tasks between the "Labor-PEs" and the "Super-PE" is managed by an autotuning system, which can modify this division according to the type of data being processed.

The autotuner in AWB-GCN dynamically distributes jobs between "Super-PE" and "Labor-PEs", according to workload and data patterns to ensure efficient processing. It evaluates, decides, and modifies job distribution to ensure peak performance.

The overall goal of AWB-GCN is to accelerate SpMM calculation by effectively allocating jobs among various processing units according to the properties of the data being processed.

QEGCN propose the feasibility of training quantized GCNs, allowing for the use of low-precision integer arithmetic during the inference phase. Fig. 9 shows the overall architecture of the QEGCN in Fig. 9(a) and in Fig. 9(b) show the detailed design of the PE block used in this architecture.

In Fig. 9(a), the FPGA memory system (BRAM and HBM memory) is exploited to optimize the memory usage. The 16 PE blocks of QEGCN are mapped onto the corresponding 16 AXI channels of the HBM memory, in order to facilitate the parallel processing of GCNs at the edge-level. As shown in Fig. 9(b), the structure of each PE block includes applying and gathering modules connected through interface FIFOs. As stated earlier, the architecture of PE follows the baseline model discussed in Section Typical architecture of a GCN-based hardware accelerator. To elaborate more, this model demonstrates the baseline architecture in Fig. 3 as considering gather as handling aggregation, and apply as performing the combination or transformation on the aggregated node features. The number of these blocks can vary based on which datatype is used in the design, which can be integers 32, 16, or 8, corresponding to creating 11, 17, and 23 numbers of "apply and gather" blocks, respectively.

Elaboration

This section is structured in three parts: (i) Methodological Considerations, Performance and Energy; (ii) discussion on limitations; (iii) overall future directions.

Comparing the performance and energy efficiency of GCN accelerators

Methodological considerations, performance and efficiency

Before proceeding with a performance comparison, some observations regarding the literature review presented in Section Overview of existing solutions are necessary. Firstly, conducting a numerical comparison among the analyzed GCN implementations is challenging due to the absence of a standardized baseline implementation and the lack of a well-defined GCN benchmark suite. In addition, GCN models and datasets may involve various settings and configurations, making it difficult to use a common baseline for a fair comparison. Therefore we developed the following methodology.

As baseline for comparing the inference performance, we have selected the Cora dataset, which is widely used in the literature. Additionally, we have chosen the HyGCN as a baseline model for the speedup comparison, as done in most of the works, since HyGCN is one of the first successful hardware implementations.

There are also other hyperparameters that are tuned in the implementations. For example, HyGCN did not use the original number of h of hidden layers ($h = 32$ as in [20]), but use $h = 128$. In AWB-GCN [1], this is observed and the two configurations of $h = 32$ and $h = 128$ are compared, thus providing interesting insights related to the performance and energy efficiency in the two cases. In particular, when $h = 32$ AWB obtains a latency of 2.3 μs and energy efficiency 3080 graphs/J compared to the case of $h = 128$ in which AWB obtains a latency of 17 *mu*s (speedup 1.24 versus HyGCN) and energy efficiency 439 graph/J. A similar situation is observed in I-GCN [5], therefore we empirically derived this difference in the latency of a factor of about 7 between the case of Boost-GCN, FlowGNN, FP-GNN, H-GCN in order to extend the comparison to cases where $h = 128$. In the case of GCoD the speedup is derived, from fig. 9 of the original paper [16] and in other cases the numbers of inference speedup or latency are derived either directly from re-implementations carried by other works.

Inference latency speedup

Considering those works that are performing better than HyGCN in Fig. 10, the reasons of the better results can be explained as follows. AWB-GCN is one of the first works that systematically consider the problem of localizing and balancing the computations of real-world graphs, i.e. those graphs that have few nodes with many connections and many nodes with few connections (see also [28, 63] for specific discussion and FPGA optimizations about such graphs that are following the *power-law*). I-GCN achieves notable speedups by enhancing data locality and minimizing off-chip memory access through the innovative "islandization" technique. Other approaches, such as SnF, EnGN, GRIP, and GCNAX leverage-optimized ASIC designs to enhance performance. In particular, SGCN outperforms all other studies by integrating data compression techniques.

FlowGNN proposes a innovative dataflow architecture and it parallelizes the operations at the edge- and node-level and internally to the aggregation engine (see Fig. 5). The node information is multicast across the data processing queues.

The best performing FPGA implementation in the set that we analyzed is reached by GCoD. The strategy of GCoD consists in pre-subdividing the graph partitions in denser and sparser, while the hardware processes them with appropriate specialized engines. On average the authors claim a speedup of 7.8x compared to HyGCN on-average,

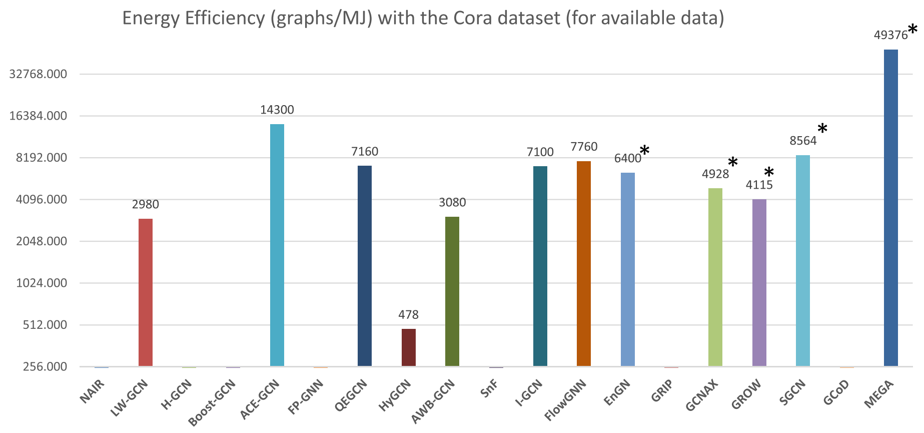

Fig. 11 Energy Efficiency comparison among GCN architectures when using the Cora dataset. FPGA-ready designs that are evaluated by using ASIC tools are marked with an asterisk

although with smaller dataset such as Core can reach about 24x. Sparsification and quantization are also providing part of this speedup.

It should also be noticed that the SnF, EnGN, GRIP, GCNAX, SGCN, GROW, MEGA implementations, while can be accommodated in an FPGA, have been evaluated by using ASIC development tools with quite aggressive technology node. This means that their superiority must be weighted compared to purely FPGA based evaluations such as FlowGNN and GCoD.

MEGA and GROW achieve the best speedup among this last category. Both design address the problem of the bottleneck of communication between the computational part and the DRAM connected to the hardware implementation. GROW is focusing on memory traffic reduction techniques of the aggregation phase. MEGA combines an adaptive quantization with a tailored scheduling of the irregular access to memory also in the aggregation phase, when a sparse-matrix multiplication is typically performed.

Inference energy efficiency

In Fig. 11, we analyze the Energy Efficiency of the GCN implementations for the data available in the literature. To allow a a more readable comparison the same color and position for each GCN architecture is maintained. The Energy Efficiency here is expressed as graphs per per Joule in most of the works. Most of the works do not provide a clear estimation of the Energy Efficiency and some of them, e.g., EnGN present the operations per Watt (power efficiency).

From Fig. 11, it is interesting to note that while some implementations may perform worse than HyGCN, like ACE-GCN, they may also provide a higher energy efficiency.

When energy efficiency is more important, therefore this implementation may results more useful. In particular, ACE-GCN opportunistically exploit a novel method called "k-largest node selection", which simplifies graph learning by finding and propagating the largest features embeddings. This is achieved by a special hardware classifier that identifies centroid in the graphs. More centroids are identified and processed in parallel and better performance can be achieved.

Limitations

A major challenge hindering the stacking of multiple layers in GCNs is the over-smoothing phenomenon. Removing the ReLU function from the given expression leads to the condition $\lim_{k\to 1}\hat{\mathbf{A}}^k\mathbf{H}^{(0)} = \mathbf{H}^{(\infty)}$, indicating that each row of $\mathbf{H}^{(\infty)}$ is solely determined by the degree of the corresponding node (cf. Section Problem Definition: background and mathematical foundations), under the assumptions of graph irreducibility and aperiodicity. This demonstrates that, when too many layers are used, the model begins to lose discriminative information provided by the node features [64].

The effectiveness of GCNs diminishes significantly when there's a severe scarcity of labeled data and when the node features become increasingly difficult to differentiate with the addition of more layers. This often leads to overfitting and over-smoothing issues [65]. Overfitting happens when the model is excessively customized to the training data, capturing noise rather than the underlying distribution, which results in poor generalization to new data. Over-smoothing, on the other hand, results in the features of nodes in the graph becoming indistinguishable, thus losing the ability to effectively differentiate between nodes.

Another limitation is the scalability of GCNs with the graph size. In case of larger graphs, the computational and memory requirements grow exponentially. This is particularly problematic for large-scale graphs that are typical in real-world applications like social networks or graphs describing molecules. Efficiently scaling GCNs to handle large graphs without sacrificing performance or accuracy remains a significant challenge.

Several overly aggressive GCN optimizations may lead to issues such as anisotropy, low expressiveness, over-smoothing, and over-squashing [66]. Anisotropy refers to the uneven distribution of information across different directions in the graph, which can lead to biased or incomplete representations. Low expressiveness denotes the model's inability to recognize complex patterns and connections within the graph embeddings. Over-squashing occurs when too much information is condensed into a small number of layers or nodes, reducing the model's capacity to learn detailed representations.

Furthermore, GCNs often struggle with handling dynamic graphs where the structure changes over time. Most current GCN models assume a static graph structure, which limits their applicability in scenarios where the graph is continuously evolving, such as in real-time recommendation systems or evolving social networks.

Addressing these limitations requires innovative approaches in both algorithm design and hardware implementation. Research efforts are being directed towards developing more robust and scalable GCN models, exploring new regularization techniques to mitigate overfitting and over-smoothing, and designing specialized hardware accelerators to efficiently process large-scale and dynamic graphs.

Future research directions

FPGA-optimizations and tools for GCNs

FPGA-based accelerators offer significant potential for enhancing the performance (i.e. graphs/s) and energy efficiency (i.e. graphs/J) of GCNs, particularly in the case of edge computing. Future research directions could focus on several key areas:

- *Optimizing FPGA Architectures for GCN Models*:
  - Tailoring FPGA architectures to specific GCN models can significantly reduce computational latency and reduce energy consumption.
  - Developing methods to exploit dynamic partial reconfiguration (DPR) can permit real-time adjustments to different graph sizes.
- *High-Level Synthesis (HLS) Tools*:
  - Leveraging HLS tools can streamline the creation and deployment of GCN accelerators on FPGAs, increasing accessibility for developers and researchers.

Quantization techniques for GCNs

Quantization methods are crucial for drastically reduce the demand of resources in GCN implementations, especially for embedded systems. Future research could explore:

- *Combining Scalar and Vector Quantization*:
  - Merging the computational efficiency of integer arithmetic with the superior compression capability of vector quantization to develop efficient reduced models can allow more aggressive hardware implementations.
- *Aggressive low-bit representations*:
  - Investigating methods that use low-bit representations (such as 4-bit and 2-bit) to maintain high accuracy while integrating large GCN models into embedded devices.
- *Fully quantized GCN models*:
  - Designing fully quantized GCN models tailored for embedded systems to achieve scalable and efficient solutions.

FPGA applications for embedded systems

Despite the advantages, there is a noticeable gap in research on GCN accelerators implemented on FPGAs and used in embedded systems. Addressing this gap involves:

- *Accelerating the training phase*:
  - Focus on accelerating the training phase of GCNs, in particular dynamic graphs, to meet the needs of real-time applications.
- *Adopting Additional Techniques*:
  - Beyond quantization, exploring additional techniques such as knowledge distillation, reordering, sampling, and simplification to enhance the efficiency of hardware-based applications.

Algorithm-hardware co-design

Future research should emphasize a co-design approach where algorithms are developed in tandem with hardware to ensure optimal performance:

- *Acceleration beyond classic GCNs*:
  - Developing hardware accelerators that can efficiently handle GNN models beyond GCNs, such as Graph Attention Networks (GATs) [50], which has distinct computational requirements. Similarly, Graph Isomorphism Networks (GINs) [67] are designed to be more powerful and discriminative in distinguishing non-isomorphic graphs. However, while these techniques may pro-

vide a little accuracy improvement, they may require a substantially complex architecture.

- *Large-scale GCNs*:
  - Creating solutions for large-scale graph processing, essential for industrial applications, where single-machine processing is impractical.
- *Emerging Devices and Architectures to be integrated with FPGAs*:
  - Exploring emerging devices, such as ReRAM crossbars and processing-in-memory (PIM) architectures can also improve performance and energy savings.
  - Investigating other novel architectures and emerging devices beyond ReRAM for better performance.

Heuristic-hardware co-operation

Incorporating heuristics into hardware design can lead to more efficient GCN accelerators:

- *Heuristic-based acceleration*:
  - Utilizing heuristics to develop hardware-aware networks that optimize operations based on prior knowledge, thereby enhancing energy efficiency and processing speed.
- *Broadening heuristic applications*:
  - Expanding the use of heuristic-based approaches to a wider range of devices beyond the current focus on ReRAM crossbars.

Directional graph networks (DGNs)

Directional Graph Networks (DGNs) [66] aim to generalize the directional features of Convolutional Neural Networks (CNNs) while mitigating their limitations. FPGA implementations of DGNs, such as those on the Alveo U50, show a potential for up to 45x improvement in performance over GPUs like the RTX A6000 [68].

Augmented message passing

GCNs are part of the broader research area of Graph Neural Networks (GNNs). The promising evolution of "augmented message passing" techniques [69] for GNNs highlights the potential for improved models. The message-passing computation paradigm provides a general framework for GNN models, facilitating the development of more efficient and powerful neural network architectures [68].

By addressing these areas, future research can improve the deployment of GCNs significantly and consequently the quality of real-world applications.

However, at the current state of technology, GCN implementations for FPGA continue to provide more and more optimized performance [70–72].

Conclusions

Graph Convolutional Networks (GCNs) have been recognized as a powerful method for processing graphs, demonstrating significant promise across several domains from social networks and biological networks to financial applications and recommendation systems. This survey has provided a comprehensive overview of GCNs, discussing their fundamental concepts, algorithmic implementations, and the challenges associated with their deployment on different hardware platforms.

Our survey highlighted the substantial advancements in GCN architectures and the ongoing efforts to optimize their performance and energy efficiency. Through detailed discussions on performance-cost tradeoffs, we examined the key metrics for evaluating GCN performance, including inference latency and energy efficiency. The comparative analysis of different FPGA-based implementations (including those case in which ASIC tools are deployed for the evaluations) underscored the varying tradeoffs between performance and the needed energy, emphasizing the need for careful consideration when selecting a deployment platform.

Despite these advancements, GCNs face several limitations that hinder their widespread adoption. Over-smoothing, scalability issues, the scarcity of labeled data, and the challenges of handling dynamic graphs remain significant obstacles. Our exploration of these limitations underscored the need for innovative approaches to address these challenges, including improved algorithm designs and specialized hardware accelerators.

To mitigate these limitations and further enhance the performance of GCNs, we identified several promising research directions. Parallelization techniques, incorporating edge and graph-level features, the development of Directional Graph Networks (DGNs), augmented message passing, and optimized FPGA implementations represent key areas for future exploration. These approaches have the potential to unlock new capabilities in GCNs, making them more robust, scalable, and efficient.

In conclusion, while GCNs have already made significant strides in various applications, continued research and development are essential to overcome existing limitations and fully realize their potential. By advancing both algorithmic innovations and hardware optimizations, we can enable more effective and efficient processing of graph-structured data, opening the door for broader adoption and new applications in diverse fields.

Acknowledgements
We found useful references in Bankovic et al. [73] for the preparation of this work. We want to thank the Xilinx/AMD Heterogeneous Accelerated Compute Clusters (HACC) for their generous accelerator card donations and for providing us with the necessary resources and support to conduct our research. Additionally, we would like to thank NVIDIA for its continuous support and donation of hardware devices to our research laboratory. These contributions have been invaluable in allowing us to conduct cutting-edge research in large-scale computing.

Author contributions
The Authors equally contributed to this paper.

Funding
The European Commission partly supported this work under the AXIOM H2020 (id. 645496), TERAFLUX (id. 249013), and HiPEAC (id. 101069836) projects. This work is partly funded by the European Union - NextGenerationEU - via the PNRR

M4C2-Inv1.4 Italian Research Center on High-Performance Computing, Big-Data and Quantum Computing, cascade funding project EDGE-ME, MUR-ID: CN0000013.

Availability of data and materials
The datasets used and analyzed during the current study are available from the corresponding authors upon reasonable request.

Declarations

Ethics approval and consent to participate
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 3 May 2024 Accepted: 13 October 2024
Published online: 11 November 2024

References

1. Geng T, Li A, Shi R, Wu C, Wang T, Li Y, Haghi P, Tumeo A, Che S, Reinhardt S, Herbordt M C. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2020.
2. Tian T, Zhao L, Wang X, Qizhe W, Yuan W, Jin X. Fp-gnn: adaptive fpga accelerator for graph neural networks. Future Gener Comput Syst. 2022;136:294–310.
3. Yan M, Deng L, Hu X, Liang L, Feng Y, Ye X, Zhang Z, Fan D, Xie Y. Hygcn: a gcn accelerator with hybrid architecture. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2020.
4. Zhang B, Kannan R, Prasanna V. Boostgcn: A framework for optimizing gcn inference on fpga. In: 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2021.
5. Geng T, Wu C, Zhang Y, Tan C, Xie C, You H, Herbordt M, Lin Y, Li A. I-gcn: a graph convolutional network accelerator with runtime locality enhancement through islandization. In: MICRO-54: 54th Annual IEEE/ACM international symposium on microarchitecture, MICRO '21. ACM, 2021.
6. Zhou Z, Shi B, Zhang Z, Guan Y, Sun G, Luo G. Blockgnn: towards efficient gnn acceleration using block-circulant weight matrices. In: 2021 58th ACM/IEEE design automation conference (DAC). IEEE, 2021.
7. Sarkar R, Abi-Karam S, He Y, Sathidevi L, Hao C. Flowgnn: a dataflow architecture for real-time workload-agnostic graph neural network inference. In: 2023 IEEE international symposium on high-performance computer architecture (HPCA). IEEE, 2023.
8. José Romero H, Chao L, Pengyu W, Chuanming S, Jinyang G, Jing W, Guoyong S. Ace-gcn: a fast data-driven fpga accelerator for gcn embedding. ACM Trans Reconfigurable Technol Syst. 2021;14(4):1–23.
9. Zhang C, Geng T, Guo A, Tian J, Herbordt M, Li A, Tao D. H-gcn: a graph convolutional network accelerator on versal acap architecture. In: 2022 32nd International Conference on Field-Programmable Logic and Applications (FPL). IEEE, 2022.
10. Yuan W, Tian T, Qizhe W, Jin X. Qegcn: an fpga-based accelerator for quantized gcns with edge-level parallelism. J Syst Archit. 2022;129: 102596.
11. Tao Z, Chen W, Liang Y, Wang K, He L. Lw-gcn: a lightweight fpga-based graph convolutional network accelerator. ACM Trans Reconfigurable Technol Syst. 2022;16(1):1–19.
12. Nair GR, Suh HS, Halappanavar M, Liu F, Seo JS, Cao Y. Fpga acceleration of gcn in light of the symmetry of graph adjacency matrix. Design. Automation Test in Europe Conference Exhibition (DATE). 2023; 4:2023.
13. Yoo M, Song J, Lee J, Kim N, Kim Y, Lee J. Sgcn: exploiting compressed-sparse features in deep graph convolutional network accelerators. In: 2023 IEEE international symposium on high-performance computer architecture (HPCA). IEEE, 2023.
14. Wu C, Tao Z, Wang K, He L. Skeletongcn: A simple yet effective accelerator for gcn training. In: 2022 32nd international conference on field-programmable logic and applications (FPL). IEEE, 2022.
15. Zhao L, Wu Q, Wang X, Tian T, Wu W, Jin X. Hugraph: Acceleration of gcn training on heterogeneous fpga clusters with quantization. In: 2022 IEEE high performance extreme computing conference (HPEC). IEEE, 2022.
16. You H, Geng T, Zhang Y, Li A, Lin Y. Gcod: graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In: 2022 IEEE international symposium on high-performance computer architecture (HPCA). IEEE, 2022.
17. Auten A, Tomei M, Kumar R. Hardware acceleration of graph neural networks. In: 2020 57th ACM/IEEE design automation conference (DAC), 2020. pp 1–6.
18. Zhang B, Zeng H, Prasanna V. Hardware acceleration of large scale gcn inference. In: 2020 IEEE 31st international conference on application-specific systems, architectures and processors (ASAP). IEEE, 2020.
19. Zeng H, Prasanna V. Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms. In: Proceedings of the 2020 ACM/sigda international symposium on field-programmable gate arrays, FPGA '20. ACM, 2020.
20. Kipf TN, Welling M. Semi-supervised classification with graph convolutional networks. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net, 2017.
21. Gori M. Machine learning: a constraint-based approach. Amsterdam: Elsevier Science; 2017.
22. Zonghan W, Pan S, Chen F, Long G, Zhang C, Philip S. A comprehensive survey on graph neural networks. IEEE Trans Neural Netw Learn Syst. 2021;32(1):4–24.

23. Zhou J, Cui G, Shengding H, Zhang Z, Yang C, Liu Z, Wang L, Li C, Sun M. Graph neural networks: a review of methods and applications. AI Open. 2020;1:57–81.
24. Hamilton W L, Ying R, Leskovec J. Inductive representation learning on large graphs. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS'17, pp 1025-1035, Red Hook, NY, USA, 2017. Curran Associates Inc.
25. Fout A, Byrd J, Shariat B, Ben-Hur A. Protein interface prediction using Graph convolutional networks. In: I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, Advances in Neural Information Processing Systems, volume 30. Curran Associates, Inc., 2017.
26. Wei J, Fang Z, Yiyang G, Liu Z, Long Q, Qiao Z, Qin Y, Shen J, Sun F, Xiao Z, Yang J, Yuan J, Zhao Y, Wang Y, Luo X, Zhang M. A comprehensive survey on deep graph representation learning. Neural Netw. 2024;173:106207.
27. Lin YC, Zhang B, Prasanna V. 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, FPGA '22. ACM, 2022.
28. Amin S, Marco B, Marco P, Wayne L, Georgi G, Roberto G. Distributed large-scale graph processing on fpgas. J Big Data. 2023;10(1):95.
29. Sun G, Yan M, Wang D, Li H, Li W, Ye X, Fan D, Xie Y. Multi-node acceleration for large-scale gcns. IEEE Trans Comput. 2022;71:1–12.
30. Dai Y, Zhang Y, Tang X. Cegma: coordinated elastic graph matching acceleration for graph matching networks. In: 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2023.
31. Cai Z, Yan X, Wu Y, Ma K, Cheng J, Yu F. Dgcl. Proceedings of the Sixteenth European Conference on Computer Systems, 2021;4.
32. Zhou H, Zhang B, Kannan R, Prasanna V, Busart Carl. Model-architecture co-design for high performance temporal gnn inference on fpga. In: 2022 IEEE international parallel and distributed processing symposium (IPDPS). IEEE, 2022.
33. Zhang B, Zeng H, Prasanna V. Accelerating large scale gcn inference on fpga. In: 2020 IEEE 28th annual international symposium on field-programmable custom computing machines (FCCM). IEEE, 2020.
34. Lecun Y, Boser B, Denker JS, Henderson D, Howard RE, Hubbard W, Jackel LD. Handwritten digit recognition with a back-propagation network. In: Touretzky DS, editor. Advances in neural information processing systems 2. Burlington: Morgan Kaufmann; 1990. p. 396–404.
35. Lecun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 1998;86.
36. Li S, Tao Y, Tang E, Xie T, Chen R. A survey of field programmable gate array (fpga)-based graph convolutional neural network accelerators: challenges and opportunities. PeerJ Comput Sci. 2022;8:e1166.
37. Demirci GV, Haldar A, Ferhatosmanoglu H. Scalable graph convolutional network training on distributed-memory systems. Proc VLDB Endow. 2022;16(4):711–24.
38. Kazi A, Farghadani S. and Nassir Navab. Ia-gcn: Interpretable attention based graph convolutional network for disease prediction. arXiv preprint; 2021.
39. Yoo M, Song J, Lee H, Lee J, Kim N, Kim Y, Lee J. Slice-and-forge: Making better use of caches for graph convolutional network accelerators. In: Proceedings of the international conference on parallel architectures and compilation techniques, PACT '22, page 40-53, New York, NY, USA, 2023. Association for Computing Machinery.
40. Li Q, Han Z, Wu XM. Deeper insights into graph convolutional networks for semi-supervised learning. *arXiv*, 2018.
41. Zhou J, Cui G, Shengding H, Zhang Z, Yang C, Liu Z, Wang L, Li C, Sun M. Graph neural networks: a review of methods and applications. AI Open. 2020;1:57–81.
42. Garg R, Qin E, Martinez F M, Guirado R, Jain A, Abadal S, Abellan J L, Acacio M E, Alarcon E, Rajamanickam S, Krishna T. A taxonomy for classification and comparison of dataflows for gnn accelerators. Technical Report SANDIA, 2021;3.
43. Abadal S, Jain A, Guirado R, López-Alonso J, Alarcón E. Computing graph neural networks: a survey from algorithms to accelerators. ACM Comput Surv. 2021;54(9):1–38.
44. Abadal S, Jain A, Guirado R, López-Alonso J, Alarcón E. Computing graph neural networks: a survey from algorithms to accelerators. ACM Comput Surv. 2021;54(9):1–38.
45. Shuman DI, Narang SK, Frossard P, Ortega A, Vandergheynst P. The emerging field of signal processing on graphs: extending high-dimensional data analysis to networks and other irregular domains. IEEE Signal Process Mag. 2013;30(3):83–98.
46. Blagojević V, Bojić D, Bojović M, Cvetanović M, Đorđević J, Đurđević Đ, Furlan B, Gajin S, Jovanović Z, Milićev D, Milutinović V, Nikolić B, Protić J, Punt M, Radivojević Z, Stanisavljević Ž, Stojanović S, Tartalja I, Tomašević M, Vuletić P. A systematic approach to generation of new ideas for PhD research in computing. Amsterdam: Elsevier; 2017. p. 1–31.
47. Li G, Muller M, Thabet A, Ghanem B. Deepgcns: can gcns go as deep as cnns? In: The IEEE international conference on computer vision (ICCV), 2019.
48. Yimeng M, Frederik W, Guy W. Scattering gcn: overcoming oversmoothness in graph convolutional networks. In: Larochelle H, Ranzato M, Hadsell R, Balcan MF, Lin H, editors. Advances in neural information processing systems. Glasgow: Curran Associates Inc; 2020. p. 14498–508.
49. Wu F, Souza A, Zhang T, Fifty C, Yu T, Weinberger K. Simplifying graph convolutional networks. In: Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, Proceedings of the 36th international conference on machine learning, volume 97 of Proceedings of Machine Learning Research, pages 6861–6871. PMLR, 2019;09–15.
50. Veličković P, Cucurull G, Casanova A, Romero A, Liò P, Bengio Y. Graph attention networks. In: International Conference on Learning Representations, 2018.
51. Zeng H, Zhou H, Srivastava A, Kannan R, Prasanna V. Graphsaint: graph sampling based inductive learning method. In: international conference on learning representations, 2020.
52. Chen J, Ma T, Xiao C. FastGCN: fast learning with graph convolutional networks via importance sampling. In: international conference on learning representations, 2018.
53. Chiang WL, Liu X, Si S, Li Y, Bengio S, Hsieh CJ. 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, KDD '19, page 257-266, New York, NY, USA, 2019. Association for Computing Machinery.

54. Li J, Louri A, Karanth A, Bunescu R. Gcnax: a flexible and energy-efficient accelerator for graph convolutional neural networks. In: 2021 IEEE international symposium on high-performance computer architecture (hpca). ieee, 2021.
55. Zhang B, Zeng H, Prasanna VK. Graphagile: an fpga-based overlay accelerator for low-latency gnn inference. IEEE Trans Parallel Distrib Syst. 2023;34(9):2580–97.
56. Sohrabizadeh A, Chi Y, Cong J. Streamgcn: accelerating graph convolutional networks with streaming processing. In: 2022 IEEE custom integrated circuits conference (CICC). IEEE, 2022.
57. Liang S, Wang Y, Liu C, He L, Li H, Dawen X, Li X. Engn: a high-throughput and energy-efficient accelerator for large graph neural networks. IEEE Trans Comput. 2021;70(9):1511–25.
58. Kiningham K, Levis P, Ré C. Grip: a graph neural network accelerator architecture. IEEE Trans Comput. 2023;72(4):914–25.
59. Hwang R, Kang M, Lee J, Kam D, Lee Y, Rhu. Minsoo Grow: A row-stationary sparse-dense gemm accelerator for memory-efficient graph convolutional neural networks. In: 2023 IEEE international symposium on high-performance computer architecture (HPCA). IEEE, 2023.
60. Zhu Z, Li F, Li G, Liu Z, Mo Z, Hu Q, Liang X, Cheng J. Mega: a memory-efficient gnn accelerator exploiting degree-aware mixed-precision quantization. In: 2024 IEEE international symposium on high-performance computer architecture (HPCA). IEEE, 2024.
61. Kiningham K, Levis P, Ré Christopher. Greta: Hardware optimized graph processing for gnns. In: Proceedings of the Workshop on Resource-Constrained Machine Learning (ReCoML 2020), 2020.
62. Gustavson FG. Two fast algorithms for sparse matrices: multiplication and permuted transposition. ACM Trans Math Softw. 1978;4:9.
63. Sahebi A, Procaccini M, Giorgi R. Hashgrid: an optimized architecture for accelerating graph computing on fpgas. Future Gener Comput Syst. 2025;162:107497.
64. Chien E, Peng J, Li P, Milenkovic O. Adaptive universal generalized pagerank graph neural network. In: International conference on learning representations, 2021.
65. Yang X, Wei K, Deng C. Csc-gcn: contrastive semantic calibration for graph convolution network. J Inf Intell. 2023;1(4):295–307.
66. Beaini D, Passaro S, Létourneau Vincent, Hamilton W, Corso G, Lió Pietro. Directional graph networks. In: Marina M, Tong Z, editors, Proceedings of the 38th International Conference on Machine Learning, volume 139 of Proceedings of Machine Learning Research, pages 748–758. PMLR, 2021;18–24.
67. Xu K, Hu W, Leskovec J, Jegelka S. How powerful are graph neural networks? In: International conference on learning representations, 2019.
68. Sarkar R, Hao C. A generic fpga accelerator framework for ultra-fast gnn inference.
69. Cković Petar V. Message passing all the way up. In: ICLR 2022 Workshop on Geometrical and Topological Representation Learning, 2022.
70. Zhao C, Faber C J, Chamberlain R D, Zhang X. Hlperf: demystifying the performance of hls-based graph neural networks with dataflow architectures. ACM transactions on reconfigurable technology and systems, 2024.
71. Wang R, Li S, Tang E, Lan S, Liu Y, Yang J, Huang S, Hailong H. Sh-gat: software-hardware co-design for accelerating graph attention networks on fpga. Electron Res Arch. 2024;32(4):2310–22.
72. Que Z, Fan H, Loo M, Li H, Blott M, Pierini M, Tapper A, Luk W. Ll-gnn: low latency graph neural networks on fpgas for high energy physics. ACM Trans Embed Comput Syst. 2024;23(2):1–28.
73. ...Milan Banković, Vladimir Filipović, Jelena Graovac, Jelena Hadži-Purić, Hurson Ali R, Aleksandar Kartelj, Jovana Kovačević, Nenad Korolija, Miloš Kotlar, Krdžavac Nenad B, Filip Marić, Saša Malkov, Veljko Milutinović, Nenad Mitić, Stefan Mišković, Mladen Nikolić, Gordana Pavlović-Lažetić, Danijela Simić, Djurdjević Stojanović, Stanković Staša Vujičić, Janičić Milena Vujošević, Sana Miodrag Živković. Teaching graduate students how to review research articles and respond to reviewer comments. Amsterdam: Elsevier; 2020. p. 1–63.

Publisher's Note