GCoD: Graph Convolutional Network Acceleration via Dedicated Algorithm and Accelerator Co-Design

Haoran You*†, Tong Geng*‡, Yongan Zhang†, Ang Li‡ and Yingyan Lin†
†*Rice University, Houston, TX*
‡*Pacific Northwest National Laboratory, Richland, WA*
{*haoran.you, yz87, yingyan.lin*}*@rice.edu*, {*tong.geng, ang.li*}*@pnnl.gov*

***Abstract*—Graph Convolutional Networks (GCNs) have emerged as the state-of-the-art graph learning model. However, it can be notoriously challenging to inference GCNs over large graph datasets, limiting their application to large real-world graphs and hindering the exploration of deeper and more sophisticated GCN graphs. This is because real-world graphs can be extremely large and sparse. Furthermore, the node degree of GCNs tends to follow the power-law distribution and therefore have highly irregular adjacency matrices, resulting in prohibitive inefficiencies in both data processing and movement and thus substantially limiting the achievable GCN acceleration efficiency. To this end, this paper proposes a GCN algorithm and accelerator Co-Design framework dubbed GCoD which can largely alleviate the aforementioned GCN irregularity and boost GCNs' inference efficiency. Specifically, on the algorithm level, GCoD integrates a split and conquer GCN training strategy that polarizes the graphs to be either denser or sparser in local neighborhoods without compromising the model accuracy, resulting in graph adjacency matrices that (mostly) have merely two levels of workload and enjoys largely enhanced regularity and thus ease of acceleration. On the hardware level, we further develop a dedicated two-pronged accelerator with a separated engine to process each of the aforementioned denser and sparser workloads, further boosting the overall utilization and acceleration efficiency. Extensive experiments and ablation studies validate that our GCoD consistently reduces the number of off-chip accesses, leading to speedups 15286×, 294×, 7.8×, and 2.5× as compared to CPUs, GPUs, and prior-art GCN accelerators including HyGCN and AWB-GCN, respectively, while maintaining or even improving the task accuracy. Additionally, we visualize GCoD trained graph adjacency matrices for a better understanding of its advantages.**

***Keywords*-GCNs; algorithm and accelerator co-design;**

I. Introduction

The recent breakthrough achieved by deep learning has motivated growing demands for deep learning powered intelligence in many daily life devices featuring contrained resources and a small form factor [26], [27]. For example, we have recently witnessed the tremendously increased excitement towards Graph Convolutional Networks (GCNs), which has achieved state-of-the-art (SOTA) performance for graph-based learning tasks. The superior performance of GCNs largely benefits from GCNs' irregular and unrestricted neighborhood connections via two primary execution phases: aggregation and combination, where the former maintains most graph processing behaviors and the latter acts more like neural networks. Specifically, during the aggregation phase, for each node in a graph, GCNs first aggregate all its neighbor nodes' features, which heavily relies on the graph structure that is inherently random and sparse; during the combination phase, GCNs transform the aggregated features through (hierarchical) feed-forward propagation to update the feature of the given node. In parallel, recent breakthroughs of GCNs have ignited an explosive interest in investigating GCNs for numerous real-world applications, including accurate advertisement in E-commerce [43] and electric grid cascading failure analysis [28]. Many of them impose stringent latency/throughput constraints, e.g., real-time decision-making. The promising performance and the potential exciting applications of GCNs come with prohibitive challenges that limit their applications to large real-world graphs and hinder the exploration of deeper and more sophisticated GCN graphs. First, graphs (or graph data), especially real-world ones, are often extraordinarily large and irregular as exacerbated by their intertwined complex neighbor connections, e.g., there are a total of 232,965 nodes in the Reddit graph with each node having about 50 neighbors [18]. One outcome is that GCNs tend to follow the power-law distribution and therefore have highly irregular adjacency matrices resulting in prohibitive inefficiencies in both data processing and movement which substantially limits the achievable GCN acceleration efficiency. Second, the dimension of GCNs' node feature vectors can be very high, e.g., each node in the Citeseer graph has 3703 features, which can lead to paramount processing costs in the combination phrase. As an illustrative example, a 2-layer GCN requires 19G FLOPs (FLOPs: floating-point operations) to process the Reddit graph [33], resulting in a latency of 2.94E5 milliseconds when executed on an Intel Xeon E5-2680 CPU platform [13]. Such a graph inference costs 2× FLOPs and 5000× latency of a 50-layer DNN, ResNet-50, the inference of which on ImageNet requires only 8G FLOPs [5] and a latency of less than 50 milliseconds [3].

To alleviate the aforementioned challenges for unleashing

* denotes equal contribution

many of GCNs' exciting applications, pioneering works have explored from either the algorithm or hardware level. From the algorithm level, commonly used compression techniques have been applied to GCNs, such as GCN quantization [33] and sparsification [23]. From the hardware level, most GCN accelerators aim to design innovative micro-architectures and dataflows to boost the acceleration efficiency of the irregular aggregation phase driven by the fact that the acceleration bottleneck of the associated highly sparse and irregular adjacency matrices. For example, AWB-GCN [13] leverages three auto-tuning techniques to dynamically balance the workload for all processing elements (PEs) to boost the efficiency. HyGCN [42] proposes a window sliding method to improve the locality of non-zero elements in GCNs' adjacency matrices, and leverages intra-vertex/node parallelism for aggregation and weight reuses for combination, respectively.

Despite their impressive performance, GCNs' acceleration efficiency is still limiting, impeding the unfolding of GCNs' great potential in many real-world applications. In this work, we advocate GCN algorithm and accelerator co-design, and make the following contributions:

- We propose a **G**CN algorithm and accelerator **Co**-**D**esign framework dubbed GCoD which alleviate the aforementioned irregularity of GCN inference at different granularities and largely resolve the bottleneck inefficiency of GCN computing by harmonizing both algorithm and accelerator innovations. To the best of our knowledge, GCoD is the first co-design framework dedicated for efficient GCN acceleration, opening up an exciting perspective for exploring much more efficient GCN solutions.
- On the algorithm level, GCoD integrates a split and conquer training strategy to polarize the graphs to be either denser or sparser in local neighborhoods without compromising the model accuracy, resulting in adjacency matrices that have two levels of workload and enjoys largely enhanced regularity and thus ease of acceleration. In this way, GCNs still preserve large degrees of irregularity (and thus high inference accuracy) while enabling regular data accesses and processes within each workload, favoring hardware efficiency and overall utilization.
- On the hardware level, GCoD integrates a dedicated two-pronged accelerator to leverage GCoD algorithm's resulting graph adjacency matrices for further boosting the acceleration efficiency. Specifically, one branch incorporates a chunk-based micro-architecture to accelerate the polarized denser subgraphs with regular/denser patterns and balanced workloads; while the other branch accelerates (mostly on-chip) the remaining irregular/sparser but largely reduced sparser workloads (a small portion of non-zeros, e.g., 30% in Cora). Results of the two branches are then aggregated without conflicts.
- Benchmarking experiments and ablation studies on five GCN models and six graph datasets consistently validate the effectiveness of our GCoD framework, e.g., GCoD leads to 15286×, 294×, 7.8×, and 2.5× speedups over CPUs, GPUs, and the existing SOTA GCN accelerators including HyGCN and AWB-GCN, respectively, while maintaining the same or an even better accuracy.

II. Related Works

Graph Convolutional Networks (GCNs). GCNs have amazed us for processing non-Euclidean and irregular data structures [49]. Recently developed GCNs can be categorized into two groups: spectral and spatial methods. Specifically, spectral methods [21], [29] model the representation in the graph Fourier transform domain based on eigen-decomposition, which are time-consuming and usually handle the whole graph simultaneously, making it difficult to parallel or scale to large graphs [11], [38]. On the other hand, spatial approaches [14], [32], which directly perform the convolution in the graph domain by aggregating the neighbor nodes' information, have rapidly developed recently. To further improve the performance of spatial GCNs, Veličković et al. [36] introduces the attention mechanism to select information which is relatively critical from all inputs; Zeng et al. [48] proposes mini-batch training to improve GCNs' scalability of handling large graphs; and Xu et al. [41] theoretically formalizes an upper bound for the expressiveness of GCNs. GCoD's innovation is general and thus can be applied on top of different GCN algorithms to boost their hardware acceleration efficiency.

GCN Compression. The prohibitive complexity and powerful performance of GCNs have motivated growing interest in GCN compression. For instance, Tailor et al. [33] for the first time shows the feasibility of adopting 8-bit integer arithmetic representation for GCN inference without sacrificing the classification accuracy; Two concurrent pruning works [23], [51] aim to sparsify the graph adjacency matrices; and Ying et al. [44] proposes a DiffPool layer to reduce the size of GCN graphs by clustering similar nodes during training and inference. Our GCoD's split and conquer training algorithm explores from a new perspective by enforcing GCNs to naturally present the desired patterns that are potentially hardware friendly and efficient.

Graph Reordering. Graph processing applications are notorious for exhibiting extreme irregularity. Prior techniques on graph reordering [17], [1], [4] exploit structural properties of real-world graphs to enhance locality, *after training* the graph. Different from existing reordering works, our GCoD algorithm is dedicated for hardware-friendly GCN inference and integrates both pruning and reordering *during GCN training* to enforce polarization and hardware-aware sparsification. As such, the resulting GCNs enjoy a higher degree of data locality while maintaining or even improving the accuracy. Specifically, our GCoD algorithm leads to a SOTA graph pruning ratio (10% edge removal

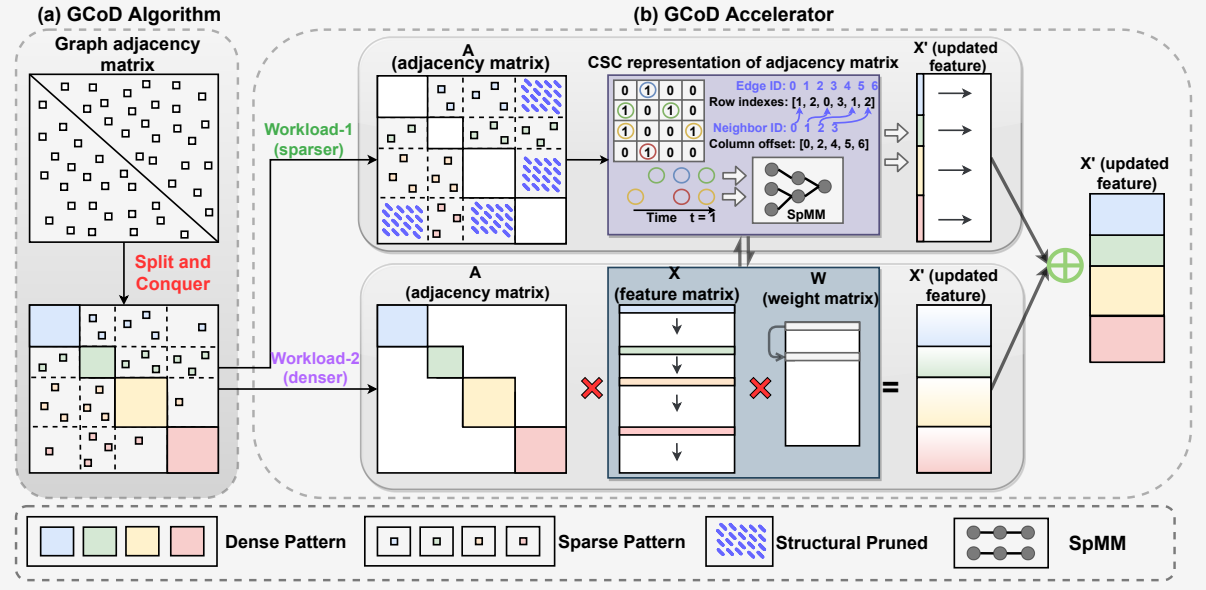

Figure 1: An overview of the proposed GCoD, an algorithm and accelerator co-design framework dedicated to GCN acceleration.

[23]) without accuracy degradation and two distinct workloads that are explicitly leveraged by our GCoD accelerator.

GCN Inference Acceleration. The ultra-sparsity in graph adjacency matrices (e.g., Pubmed dataset has 99.989% sparsity vs. sparse DNNs have 88.9 ~ 92.3% sparsity [15]) requires dynamic and irregular data accesses for GCNs' feature aggregation, which has a different execution pattern from DNN accelerators and thus motivates dedicated GCNs accelerators [2], [25]. HyGCN [42] characterizes the hybrid execution patterns for exploring both the intra/inter-vertex parallelisms to handle the irregularity in the aggregation phase and reusability in the combination phase, respectively. Later, AWB-GCN [13] identifies the workload imbalance problem in the aggregation phase, the non-zero values (i.e., connected neighbors) in adjacency matrices are regionally clustered, and proposes autotuning workload balancing techniques to alleviate the runtime imbalance. Another trend is to summarize the design space of the dataflow and micro-architecture optimization in GCN accelerators [12], and provide automated framework to generate suitable hardware for the given GCN applications [24], [50]. For example, G-CoS [50] develops the first co-search framework that can automatically search for the matched GNN structures and accelerators to maximize both task accuracy and acceleration efficiency. In contrast, Our GCoD explores dedicated algorithm-accelerator co-design for GCN inference in order to further boost the overall utilization and acceleration efficiency thanks to the enforced regularity.

III. GCoD: Motivation & Overview

Why GCN Inference Is Inefficient. There exists a fundamental dilemma associated with GCN inference acceleration: To accelerate GCN inference, the irregularity of GCNs' adjacency matrices need to be reduced, which can inevitably degrade the inference accuracy; On the other hand, maintaining GCNs' irregularity and thus their excellent accuracy can lead to extremely high hardware costs of GCN inference as demonstrated in recent works [42], [13], [25], [24], [20], [7], [12], [47]; both limiting their more extensive applications.

Why GCoD Boosts GCN Inference Efficiency. Fig. 1 shows an overview of the proposed GCoD framework, which resolves the above dilemma in a clever way. We first leverage a split and conquer training algorithm to polarize the graph to be either denser or sparser, and then design a dedicated two-pronged accelerator for separately handling each of the resulting two levels of workload. The motivating intuition is very simple: If the mass irregularity is clustered into different classes with each requiring similar data access and process patterns and being handled by a dedicated sub-accelerator, then the above dilemma can be largely resolved when dedicating one sub-accelerator to process one of the aforementioned two workloads. In this way, GCNs still preserve their advantageous large degrees of irregularity (and thus inference accuracy) while enabling mostly regular data accesses and processes within each workload, favoring potential hardware efficiency and overall utilization. Next we will introduce our GCoD algorithm and accelerator.

IV. The Proposed GCoD Algorithm

In this section, we present the proposed GCoD algorithm. Specifically, we first present the preliminaries of GCN training and graph optimization in Sec. IV-A, and then introduce the design considerations, detailed GCoD

algorithm formulation, and the early-stop efficient training pipeline in Sec. IV-B.

A. Preliminaries of GCNs and Graph Optimization

GCN Notation and Formulation. Let $\mathcal{G} = (V, E)$ represents a GCN graph, where $v_i \in V$ and $(v_i, v_j) \in E$ denote the nodes and edges, respectively; and $N = |V|$ and $M = |E|$ denote the total number of nodes and edges, respectively. The node degrees are denoted as $d = \{d_1, d_2, \cdots, d_N\}$ where d_i indicates the number of neighbors connected to the node v_i. We define D as the degree matrix whose diagonal elements are formed using d. Given the adjacency matrix A and the feature matrix $X = \{x_1, x_2, \cdots, x_N\}$ of the graph G, a two-layer GCN model [21] can then be formulated as:

$$Z = f(A, X) = \text{softmax}\left(\hat{A}\ \text{ReLU}\left(\hat{A}XW_0\right)W_1\right), \quad (1)$$

where $\hat{A}$ is a normalized version of A: $\hat{A} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$. The whole GCN inference can be viewed as two separated phases: *Aggregation* and *Combination*.

- *Aggregation*: For each node in the graph, a GCN aggregates its 1-hop neighbor nodes' feature vectors into a unified feature vector, which corresponds to the multiplication of the adjacency matrix and feature matrix $\hat{A}X$.
- *Combination*: The aggregated feature vector will be further transformed to another feature vector using an MLP network (shared between nodes) for learning better representations, which corresponds to the multiplication between the feature matrix and weight matrix, i.e., XW.

After the feature vectors update, a softmax function is applied in a row-wise manner, i.e., $softmax(x_i) = \exp(x_i)/\sum_i \exp(x_i)$ [21]. For semi-supervised multiclass classification, the loss function captures the cross-entropy errors over all labeled examples:

$$\mathcal{L}_{GCN}(W) = -\sum_{n \in \mathcal{Y}_N} \sum_{f} Y_{nf}\, ln(Z_{nf}), \quad (2)$$

where $\mathcal{Y}_N$ is the set of node indices that have labels, Y_{nf} is the ground truth label matrix, and Z_{nf} denotes the predicted possibilities of node n belonging to class f. During GCN training, W_0 and W_1 are updated via gradient descents.

Graph Optimization. The goal of graph optimization is to enforce the graph adjacency matrix for achieving desired patterns, which usually improve the graph's regularity so that the GCN models can be more hardware friendly on their target platforms. For example, graph sparsification aims to reduce the total number of edges in graphs (i.e., the size of the adjacency matrix). A SOTA graph optimization pipeline [23] is to first pretrain GCNs on their full graphs, and then optimize the graphs based on the pretrained GCNs. Note that the weights of GCNs are not updated during graph optimization, during which W is replaced with A in Eq. (2) to derive the loss function $\mathcal{L}_{GCN}(A)$. As such, the overall loss function during graph optimization can be written as [23]:

$$\mathcal{L}_{Graph}(A) = \mathcal{L}_{GCN}(A) + \mathcal{L}_{Reg}(A), \quad (3)$$

where $\mathcal{L}_{Reg}$ denotes the regularization term. Such a graph optimization enables practitioners to enforce regularized patterns in graphs for achieving more efficient GCN inference.

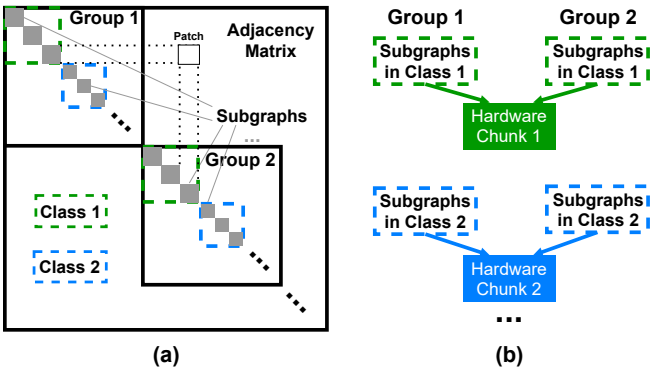

Figure 2: Illustrating (a) GCoD's defined *group*, *class*, and *subgraph* within GCNs' graph adjacency matrices, where nodes with similar degrees are categorized into the same class, each class is further divided into subgraphs with a similar number of edges, and all the subgraphs within the same class are evenly distributed into different groups, and (b) each hardware chunk (i.e., sub-accelerator) handles the same kind of classes from all the groups.

B. The Proposed GCoD Algorithm

1) GCoD: Split and Conquer Algorithm: **Split and Conquer Chunk Design.** Our GCoD algorithm alleviates the ultra-high sparsity (e.g., 99.989% in the Pubmed dataset) and irregularity in GCNs' adjacency matrices by leveraging *subgraph classification* to enforce regularity at both coarse- and fine-grained granularities, and adopt *group partitioning* to further boost the processing efficiency:

- *Subgraph Classification*: To reduce the irregularity of GCNs' graph adjacency matrices, we cluster nodes with similar degrees into the same class (different classes are denoted using dashed boxes in Fig. 2). To further achieve a finer-grained regularity within each class, we further divide each class into subgraphs with each having a similar number of edges. Therefore, subgraphs within the same class share balanced workloads and favor regular and thus efficient hardware acceleration, which are processed using one sub-accelerator (i.e., chunk) in GCoD. Sub-accelerators have unique hardware resources dedicated to handle the workload patterns of the corresponding subgraphs, and can process in parallel.
- *Group Partitioning*: We uniformly distribute subgraphs within the same class into different groups. Such group partitioning reduces the boundary connections to enforce the sparser patterns. Therefore, we can treat all of the

sparser patterns as one unique workload (w.r.t. a sub-accelerator), which simplifies hardware designs and the communication among different sub-accelerators.

The resulting patterns' irregularity in GCNs' graph adjacency matrices still maintain relatively high, while each sub-accelerator can process the same class of nodes having similar degrees and thus similar data access and process workloads, which can also be well supported by our dedicated accelerator (see Sec. V) as well as SOTA chunk-based accelerator architectures [31]. Next, we introduce GCoD training pipeline to enforce the above patterns, and its further structural removal of graph connections to increase structured sparsity.

Training Pipeline. As shown in Fig. 3, the training pipeline of GCoD can be divided into three steps: (1) pretraining GCNs on the partitioned graphs; (2) tuning (i.e., sparsify and polarize) the graphs based on the pretrained GCN model; and (3) structural sparsify graph adjacency matrices. In Steps (2)-(3), GCN retraining is needed to restore the test accuracy after tuning or sparsification, where Step (2) can be iterated several times to maximize the sparsity while maintaining the accuracy. Next, we furhter elaborate each step, including the *graph partitioning*, *sparsify and polarize*, and *structural sparsification*, of Steps (1)-(3).

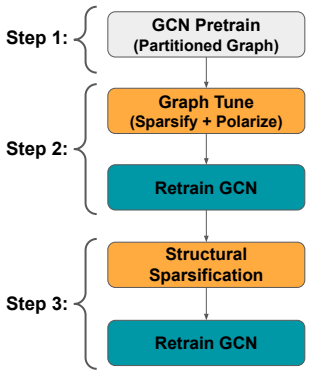

Figure 3: GCoD training flow.

- *Step 1: Graph Parititioning.* Suppose we aim to separate a graph into G groups and each group contains C subgraph classes. To partition the graph into S subgraphs, we first extract C subgraph classes $\mathcal{G}[c] = \{i \mid \hat{d}_{c-1} \leq d_i < \hat{d}_c\}$ for $1 \leq c \leq C$, with d_i representing the in-degree of the i-th node which will be divided to the c-th class, if $\hat{d}_{c-1} \leq d_i < \hat{d}_c$. Note that $\hat{d}_c$ belongs to the degree partition list which is predefined as $0 = \hat{d}_0 < \cdots < \hat{d}_C = \infty$. In this way, all nodes in the same class $\mathcal{G}[c]$ share similar degrees. Next, we use METIS [17] to partition each class $\mathcal{G}[c]$ into workload balanced subgraphs. Finally, the S subgraphs are then uniformly distributed across G groups.
- *Step 2: Sparsify and Polarize.* The goal of graph sparsification and polarization is to reduce the total number of non-zero values within the adjacency matrices (i.e., the edges in GCNs' graphs) and polarize the adjacency matrices towards denser and sparser patterns, respectively. Note that the weights of GCNs are not updated during this step. Therefore, W is replaced with A in Eq. (2) to derive the loss function $\mathcal{L}_{GCN}(A)$. In summary, the overall loss function for this step can be written as:

$$\mathcal{L}_{Graph}(A) = \mathcal{L}_{GCN}(A) + \mathcal{L}_{SP}(A) + \mathcal{L}_{Pola}(A), \quad (4)$$

where $\mathcal{L}_{SP}$ and $\mathcal{L}_{Pola}$ denote the sparse regularization and polarization terms, respectively. Ideally, $\mathcal{L}_{SP}$ will become zero if the sparsity of the graph adjacency matrices reaches the target pruning ratio (e.g., $\|A_{prune}\|_0 / \|A\|_0 \leq 1 - p$ for a given ratio of p); $\mathcal{L}_{Pola}$ equals to $1/M \cdot \sum \|i - j\|$, where M denotes the total number of non-zeros in the adjacency matrices, and i, j represent the x- and y-coordinate values of all non-zero elements within the adjacency matrices A. As the overall loss function $\mathcal{L}_{Graph}(A)$ is not differential, we follow [23] to use ADMM for minimizing the non-differential loss function using gradient descent.
- *Step 3: Structural Sparsification.* To further improve GCNs' inference efficiency, GCoD algorithm further leverages the patch-based structural sparsity within the graph adjacency matrices, where the patch definition is illustrated in Fig. 2. Specifically, GCoD prunes patches of which the number of non-zero elements is smaller than a specified threshold η, which balances the resulting sparsity and achieved GCN accuracy. In this work, η ranges from 10 to 30 for different graph datasets. As a results, the final optimized graph adjacency matrices will have some vacancies as shown in Fig. 4.

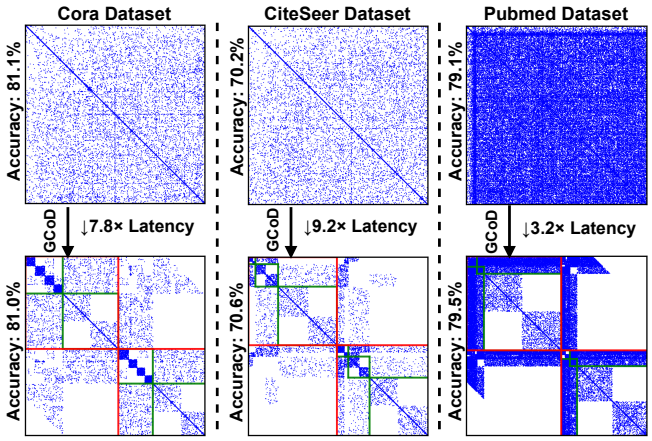

Figure 4: Visualizing three datasets' graph adjacency matrices **before and after** applying our GCoD algorithm, where green lines separate subgraph classes while red lines partition the groups. Note that non-zero dots in the matrices are enlarged for better visualization effects.

Visualization. We visualize the graph adjacency matrices before and after GCoD training in Fig. 4, which clearly demonstrates the improved regularity and the effectiveness of GCoD split and conquer training strategy: achieving on average 7.8× speedups over HyGCN [42] when evaluating on the proposed accelerator architecture in Sec. V, while maintaining or even improving the test accuracy.

2) Efficient Training Pipeline via Early-stopping: The vanilla training algorithm of GCoD enables the opportunity of boosting GCN inference efficiency yet requires a non-trivial training overhead as compared to the standard GCN training. To reduce its training overhead, we propose to first identify the winning subnetworks from the origin GCN networks at the very early training stages (e.g., 10~20 epochs over a total of 400 training epochs) following [45], [46], resulting in much improved training efficiency in both Step 1 (via early-stopping) and Steps 2-3 (via only retraining GCN subnetworks) without compromising the final accuracy. In this way, our GCoD algorithm only requires a comparable or even a lower training cost **(0.7× ~ 1.1×)** than the standard GCN training. In particular, GCoD training algorithm leads to at most 10% training overhead when evaluated on SOTA five models and six datasets. The three steps in GCoD training algorithm account for about 5%/50%/45% of the total training cost, respectively, where the dominated steps 2-3 require to retrain the GCN subnetworks from scratch.

V. The Proposed GCoD Accelerator

A. Motivation for GCoD Accelerator

Opportunity. Our proposed GCoD algorithm exhibits a great potential in alleviating the irregularity in GCNs' graph adjacency matrices. However, this potential cannot be fully exploited by existing GCN accelerators [42], [13] due to (1) the resulting two distinct workloads from GCoD algorithm, i.e., the sparser and denser branches as shown in Fig. 1; and (2) the lack of opportunities in existing GCN accelerators to dedicate for different classes (w.r.t. similar node degrees and balanced workloads within each of them) in the denser branch, and to fully leverage the reduced workloads and enforced structural sparsity in the sparser branch. As such, GCoD accelerator is motivated to take advantages of the new opportunities resulting from GCoD algorithm to further boost the acceleration efficiency.

Design Exploration. Here we first discuss the two typical designs in existing GCN accelerators for accelerating the aggregation phase, i.e., the performance bottleneck of GCN processing, and elaborate their advantages and disadvantages. To handle the dominant aggregation phase, SOTA GCN accelerators either adopt *gathered* aggregation or *distributed* aggregation. In particular, **gathered aggregation** (see Fig. 5 (a)) executes nodes in a sequential manner, where the neighbor features of each node are gathered in parallel for aggregation. The advantage is that it requires only a small buffer for handling the aggregation results thanks to the good reuses of the intermediate aggregation outputs. However, such gathered aggregation causes irregular and frequent (off-chip) accesses of the weights and features, which is often too large to be stored on-chip, due to the sparse and random distribution of non-zeros in the adjacent matrix. For example, HyGCN [42] adopts such gathered aggregation. On the other hand, the latter, i.e., **distributed aggregation** (see Fig. 5 (b)), executes nodes distributively in a parallel manner, where the neighbor features of each node are gathered sequentially for aggregation. Its advantage is that the weight features can be fully reused, because it processes the non-zero elements of the adjacent matrix in a column-wise manner and thus allows the rows of the weight matrix to be reused by all the elements of the same column in the adjacent matrix. This advantage yet comes at a cost of requiring a large buffer to hold the intermediate aggregation results, which is often too large to be stored on-chip and thus leads to frequent off-chip accesses. Furthermore, with the often varying number of neighbors for different nodes, the required off-chip memory accesses can be rather irregular and thus leads to further workload imbalance and inefficiency. For example, AWB-GCN [13] adopts such distributed aggregation.

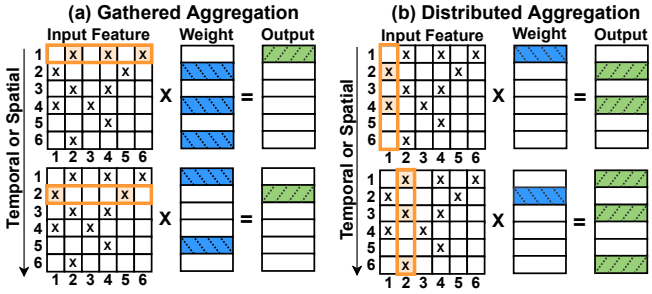

Figure 5: Illustrating the *gathered* and *distributed* aggregations.

In general, the gathered aggregation needs more off-chip bandwidth to access weights and features while the distributed aggregation needs more on-chip storage to hold the aggregation results. Given that bandwidth is more limited when processing GCNs which have a poor data locality and extreme irregularity, the distributed aggregation better favors GCN acceleration efficiency as compared to the gathered aggregation, as verified by the much improved performance offered by AWB-GCN (distributed) over HyGCN (gathered). As such, GCoD considers the distributed aggregation design, and strives to further alleviate the associated workload imbalance problem and to reduce on-chip storage requirements for much boosted GCN inference efficiency. In particular, GCoD algorithm training enforces two distinct workloads for the aggregation phase without compromising the accuracy. Each of the resulting two workloads is now better suited for overcoming distributed aggregation's disadvantages. We next present the detailed micro-architecture design in Sec. V-B.

B. GCoD Accelerator's Micro-architecture

Architecture Overview. Fig. 6 illustrates the overall micro-architecture of the GCoD accelerator. For better processing elements (PEs) utilization and reduced off-chip memory access during the performance dominant aggregation phase, GCoD accelerator consists of two separate

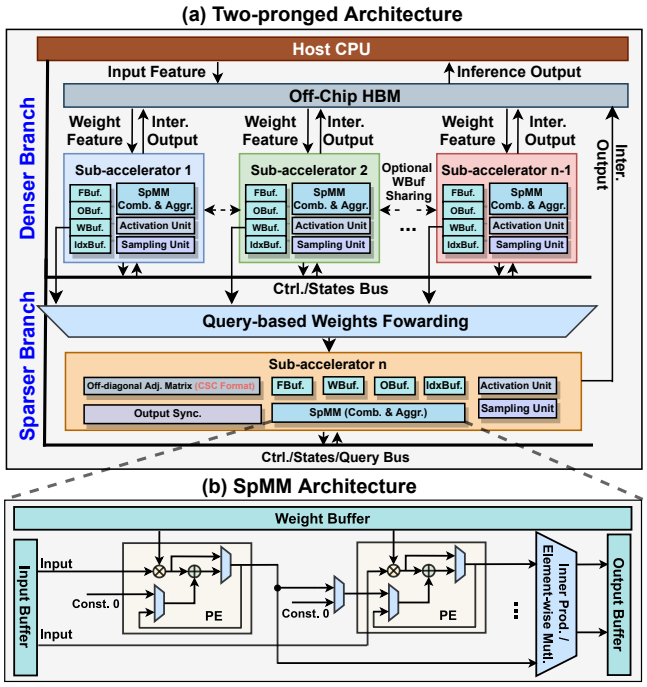

Figure 6: Illustrating the micro-architecture of GCoD accelerator.

computing branches with each dedicated to process the denser workload and sparser workload of GCoD algorithm's resulting adjacency matrices, respectively. As shown in the upper part of Fig. 6, **the Denser Branch** processes the enforced regular dense subgraphs along the diagonal line of the adjacency matrices with an array of parallel sub-accelerators to handle the more intense workload while maintaining the workload balancing through complexity proportional resource allocation among the sub-accelerators. **The Sparser Branch**, as shown in the bottom part of Fig. 6, at the same time handles the remaining irregular but significantly lightweight sparser workloads mostly *on-chip*, through the Compressed Sparse Column (CSC) input format and the proposed weight forwarding technique, largely avoiding frequent and large volume data movements from the off-chip memory. Each sub-accelerator within the branches is capable of processing the task of sparse-dense matrix multiplication (SpMM), non-linear activation, and node sampling. We term the proposed micro-architecture comprising two separate computing branches as *two-pronged architecture* hereafter. The characteristics of each branch are summarized in Tab. I. For minimal controlling overhead and interruption to the computing flow, these two branches are equipped with separate output buffers, so that their generated results can be written into the buffers in parallel and are further synchronized and accumulated to produce the final outputs, as illustrated in Fig 1. We next elaborate the design of each branch.

Denser Branch. As shown in Sec. IV-B, subgraphs from

Table I: Summarizing the characteristics of GCoD accelerator's Denser and Sparser Branches.

	Multi Chunks	On-chip Storage	Off-chip Access	Arch. Reuse	Data Reuse	Workloads
W/o GCoD	N	H	H	N	N	**Heavy & Imbalanced**
GCoD Denser	Y	L	L	Y	Y	**Balanced**
GCoD Sparser	N	H	L	Y	Y	**Light**

different classes will have different dimensions, resulting in different workload sizes when being processed. Thus, to balance the workload of processing the subgraphs from different classes and meanwhile maintain a high degree of parallelism, we adopt a chunk-based architecture for the denser branch acceleration, where sub-accelerators process the subgraphs from different classes simultaneously. To achieve balanced workload, we first estimate the workload size for the subgraphs within each class and assign each sub-accelerator to a different class. After that, given the available hardware resource budget, we allocate hardware resource to each sub-accelerator proportional to its assigned subgraphs' workload size. In particular, (1) for PEs allocation, we use the number of multiply and accumulate operations (MACs) with sparsity considered to characterize each workload's size, and assign each sub-accelerator the number of PEs proportional to the assigned subgraphs' number of MACs; (2) For on-chip memory and off-chip bandwidth allocation, we first calculate all the input/output feature maps and weights sizes when processing the subgraphs within each class, and then similarly assign memory and bandwidth to each sub-accelerator proportional to the sum of the calculated feature maps and weights sizes. As the subgraphs within each class have similar workload sizes, benefiting from GCoD algorithm's integrated subgraph classification, GCoD accelerator adopts the same sub-accelerator assigned to each class to process the subgraphs within the same class. For convenience, hereafter we term a sub-accelerator for processing workload balanced subgraphs within the same class as a "chunk", as illustrated in Fig. 6 (a). Note that the number of chunks equals to the number of classes and is obtained during GCoD algorithm's training. Throughout the densor branch, either dense or Coordinate (COO) format inputs and weights are assumed for reduced controlling overhead.

It is worth noting that such a chunk-based accelerator design naturally achieves workload balance without the necessity of on-the-fly autotuning, differentiating itself from previous SOTA accelerators, e.g., AWB-GCN [13].

Sparser Branch. Benefiting from GCoD algorithm, the off-diagonal workloads feature much reduced data and computation density (50% ~ 75%), as shown in Fig. 4. As such, GCoD accelerator handles the data movements associated with these workloads, i.e., when processing the workload in the sparser branch, with (mostly) the on-chip memory by utilizing (1) a CSC data format for the input data (i.e., entries

in the adjacency matrices) and (2) query-based weight (i.e., features to be aggregated) forwarding.

- *CSC Format Inputs*: By storing in the CSC data format, GCoD accelerator is able to fit most of the input data to the on-chip buffer, thanks to the drastically reduced adjacency matrices' density and CSC's smaller storage overhead as compared with the COO format. To be compatible with the CSC format and the subsequent weight forwarding technique, GCoD accelerator adopts the distributed aggregation design (see Fig. 5 (b)), which consumes column(s) of the inputs every clock cycle.
- *Query-based Weight Forwarding*: Because the denser and sparser branches operate in parallel, when the sparser branch is working on certain columns of the input data, it is likely that some chunks in the denser branch is working on the same columns but different rows, thus sharing the same needed rows of weights as the sparser branch. Therefore, for the weights needed during the sparser branch, instead of loading them from the off-chip memory, the sub-accelerator in the sparser branch will query the weight buffers of the corresponding chunks in the denser branch for accessing the already loaded weights. Specifically, the weight forwarding is performed on the demand of the sparser branch. The sparser branch sub-accelerator first determines which of the other sub-accelerators to query based on the queried weights' row indices which are also the adjacency matrix's column indices. By checking the predefined location of the queried sub-accelerator's index buffer, the sparser branch sub-accelerator will acquire the range of the weight data currently stored in the weight buffer. If the queried data falls in the range, its address within the weight buffer will be calculated based on the known range. Then, the sparser branch sub-accelerator will access that specific address for the queried weight data. Overall, for the sparser branch's weight, about 63% of the data will be accessed through the query-based weight forwarding. In particular, the sparser branch sub-accelerator works on multiple columns across different classes, simultaneously. The sparser branch sub-accelerator is able to finish the computation at the similar pace to all the parallel sub-accelerators in the denser branches, because of the increased sparsity and predefined resource allocation. The overall matched pace ensures the decent amount of likelihood of weight forwarding. However, because of the further temporal tiling of each class i.e., the buffer may not fit all the weights belonging to the specific class and the denser and sparser branches will not be synchronized until the end of aggregation, weight forwarding conditions will not always be satisfied. For these cases, weights will be instead loaded from the off-chip memory.

Our GCoD accelerator adopts a similar architecture design and workload allocation for processing the sparser workloads of the sparser branch as that of the denser branch, but with only one sub-accelerator. For larger graphs scaling to billion-edge levels under which the on-chip storage cannot handle the sparser workloads, our GCoD accelerator will consider fine-grained pipelines as described in Sec. V-B to separate the workloads and continue maintaining the less off-chip memory accesses benefits as compared to vanilla graphs. In addition, the additional structural sparisty (up to 10%) of GCoD algorithm's adjacency matrices leads to more columns to be entirely skipped and thus facilitates the accumulation of partial results from two branches.

Sub-accelerator Architecture. As shown in Fig. 6, to support different layer dimensions and the special operations from various GCN structures, the sub-accelerators are equipped with multiple functional units:

- **Dedicated Buffers** to favor local reuse opportunities, with sizes decided in the above resource allocation stage. The buffers are implemented in either block RAM or look-up tables depending on their sizes and required parallel read/write ports.
- **Sparse/Dense Matrix Multiplication Engine (SpMM)** which supports both dense and sparse matrix multiplication. Specifically, in the dense format, it takes vectors of inputs and weights and performs either inner products or element-wise multiplications between inputs and weights, as shown in Fig. 6 (b). When sparsity is considered, thanks to the adopted COO input format or the CSC input format (in distributed aggregation mode), only non-zero elements of the inputs and their indices are loaded for calculation. Specifically, when handling the sparse matrix multiplication in the denser branch or combination phase, we use the COO format for the adjacency/feature matrix storage and the dense format for the weight storage. As such, by nature we can access only the non-zero adjacency/feature matrix' values and their location indices to support the sparsity. The loaded elements will be piped to the PEs in Fig. 6 (b) for result calculations. On the other hand, when handling the sparse matrix multiplication in the sparser branch, we use the CSC format for the adjacency matrix for a smaller storage overhead and the dense format for the transformed (combined) features. Column(s) of the non-zero adjacency matrix elements will be loaded at the same time. With the dataflow restricted to the distributed fashion as in Fig. 5 (b), we can produce the rows of outputs only corresponding to the non-zero elements in each adjacency matrix's column at a time to fully leverage the sparsity.
- **Element-wise Activation Units** for the non-linear activation operations. Specifically, we use gating modules for the ReLU and lookup tables to estimate other non-linear activation functions.
- **Sampling Units** to schedule the node sampling. Specifically, we implement a linear shift register to randomly pick from non-zero elements from the adjacency matrices' columns.

The overall micro-architecture is reused for GCN aggre-

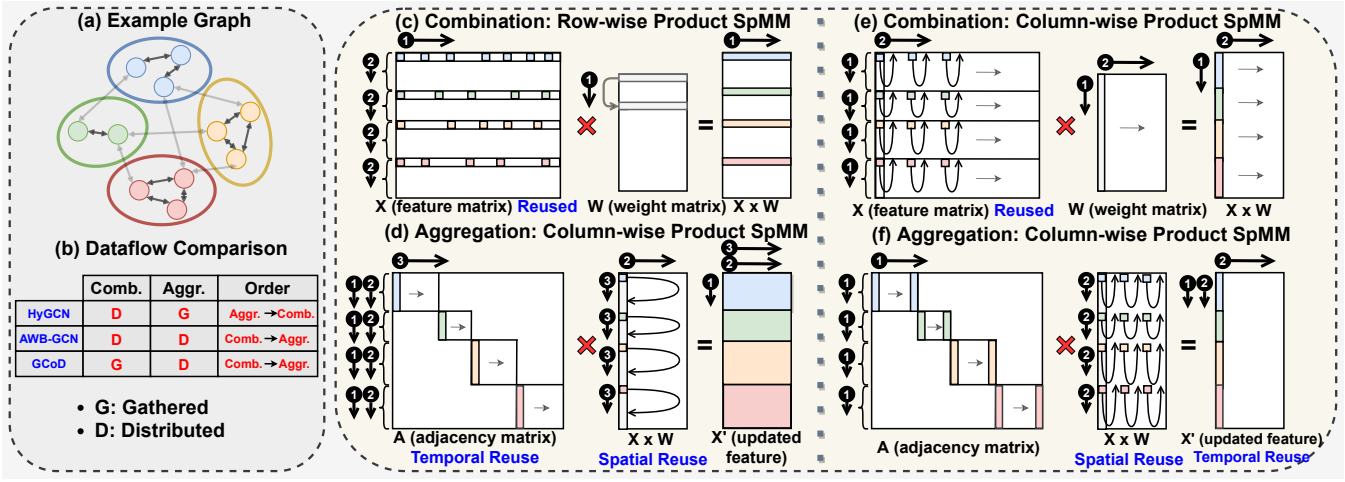

Figure 7: An illustration of the efficiency- and resource-aware pipelines for various data reuses within our GCoD accelerator, where the execution order is numbered in each sub-figure.

gation and combination, as both involve mostly matrix multiplications. The sub-accelerator in the sparser branch has an additional output synchronization module to combine its outputs with the denser branch's outputs. To support large GCNs, computation tiling can be achieved by scheduling the input/weight vectors loaded to the sub-accelerator, without modifying the underlying hardware. Each sub-accelerator communicates with an off-chip High Bandwidth Memory (HBM) through direct memory access to increase the access efficiency.

Efficiency- or Resource-aware Pipeline. Distributed aggregation helps reduce off-chip memory accesses. GCoD further enables the intermediate results (i.e., one row of XW) to be directly utilized by aggregation through performing combination in a row-wise gathered manner. Such inter-phase pipeline can largely boost the efficiency and thus is termed as *efficiency-aware pipeline*, as illustrated in Fig. 7 (c) and (d). However, the above efficiency gain is achieved at a cost of requiring a large on-chip accumulation buffer for storing the aggregation results, otherwise the results have to be transferred back to the off-chip memory. Benefiting from GCoD's split and conquer training, such an on-chip large buffer can be alleviated but is still required for processing billion-edge large graphs. To tackle this problem, we proposed a *resource-aware pipeline* where outputs are also temporally reused so that only one column of aggregation outputs needed to be stored on-chip, making it more suitable for handling large graphs. We summarize these two pipelines' characteristics in Tab. II and compare GCoD's dataflow with prior works in Fig. 7 (b). Next, we elaborate data reuses and dataflows in each pipeline.

The core idea of the efficiency-aware pipeline is to perform combination in a row-wise gathered manner so that the intermediate results (i.e., one row of XW) can be directly utilized by aggregation. During combination, GCNs perform SpMM between the features X and the weights W. Since the dense W is much smaller than X (e.g., 300KB vs. 534MB for Reddit), it can be fully stored on-chip and accessed multiple times upon request without compromising the efficiency. As such, a row-wise product order is adopted (see Fig. 7 (c)). Within each sub-accelerator, each non-zero element in X needs to multiply an entire row of W, and once all elements at the same row of X are fully processed, an entire and dense row of XW is calculated. During aggregation, for reducing the off-chip memory accesses of the adjacency matrix A, as shown in Fig. 7 (d), our GCoD accelerator adopts distributed aggregation to largely reuse the resulting matrix XW, which is both dense and large (e.g., 114MB for Reddit). Within each sub-accelerator, each element of the resulting row of XW multiplies all the non-zeros in the corresponding column of matrix A so that XW can be fully and spatially reused and A can be temporally reused by all elements in the row of XW.

Table II: Comparison of two inter-phase pipelines, where RW and CW represent row- and column-wise products.

Inter-Phase	Comb. SpMM	Aggr. SpMM	On-chip Storage	Off-chip Access	Data Reuse	Fit for Graphs
Efficiency-aware	RW	CW	H	L	X, XW, A	Medium
Resource-aware	CW	CW	L	L	X, XW, X′	Large

For resource-aware pipeline, During combination, as illustrated in Fig. 7 (e), the data reuse pattern remains the same as the efficiency-aware pipeline (i.e., reuse A) while the execution order changes from row-wise to column-wise so that the intermediate results will be one column of XW. During aggregation, as illustrated in Fig. 7 (f), within each sub-accelerator, each element of the resulting column of XW multiplies all non-zeros of A so that not only XW but also the outputs can be fully reused. In this way, we only need to store one column of aggregation outputs to the output buffer.

Reconfigurability. To support the potential task change

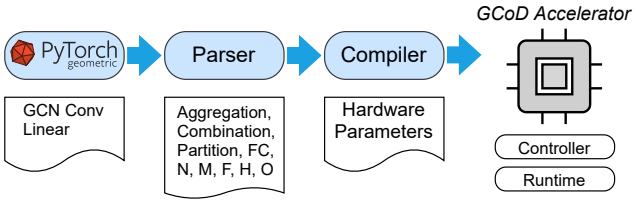

Figure 8: The software-hardware interface pipeline for GCoD.

after deployment, the proposed GCoD is equiped with a low-cost hardware reconfigurable strategy for use during the hardware compilation process and this compilation cost is a one-time effort for each task. Specifically, a series of C/Verilog based code templates are developed to form the overall hardware architectures with parameterizeable attributes, e.g., number of chunks, PEs, and buffers' sizes. After that, the given GCN will be passed through a network parser to feed the hardware compiler with layer dimensions, e.g., feature size, as illustrated in Fig. 8. The hardware compiler will fill in the parameterized attributes in the previous code templates. The configured hardware architecture will be sent to the platform software, e.g., Vivado [40], to generate the bitstream for on board deployment. The reconfigurable process cost is amortized across the entire lifetime of each task.

VI. Experiment Results

In this section, we present a thorough evaluation of the proposed GCoD framework, including the overall benchmark with CPUs/GPUs and SOTA GCN accelerators in Sec. VI-B, and the evaluation and ablation studies of GCoD algorithm and accelerator in Sec. VI-C and Sec. VI-D, respectively.

A. Experiment Settings

Models, Datasets, and Training Settings. Our evaluation considers **five GCN algorithms**, including three representative full-batch training GCN algorithms (i.e., GCN [21], GAT [35], and GIN [41]), one mini-batch training GCN algorithm (i.e., GraphSAGE [14]), and one deep ResGCN [22], and **six graph datasets**, including three citation graph datasets (Cora, CiteSeer, and Pumbed) [30], the one knowledge graph (NELL) [6], and the two large scale datasets (Obgn-ArXiv from Open Graph Benchmark (OGB) [16] and Reddit post dataset [14]), respectively. The specifications of the aforementioned five GCN models are summarized in Tab. IV, from which we can see that the adopted GCNs and GINs consist of 16 hidden units for the three citation graphs and 64 hidden units for the NELL and Reddit graphs, following a SOTA GCN accelerator [13]; the GAT models consist of 8 hidden units and 8 heads; the GraphSAGE models adopt two layers and the same hidden dimensions as GCNs, with a neighborhood sample size of 25 and 10, respectively, following the basic settings in [9], [14]; and deep ResGCNs adopt 28 layers with 128 hidden units, following the settings in [22]. We train all the above GCN models for 400 epochs using an Adam optimizer [19] with a learning rate of 0.01. The statistics of these six datasets are summarized in Tab. III. We follow the same dataset splits as described in [21], [14], [16].

Table III: A summary of the adopted graph dataset statistics.

Dataset	Nodes	Edges	Features	Classes	Storage
Cora	2,708	5,429	1,433	7	15 MB
Citeseer	3,312	4,372	3,703	6	47MB
Pubmed	19,717	44,338	500	3	38MB
NELL	65,755	266,144	5,414	210	1.3GB
Ogbn-ArXiv	169,343	1,166,243	128	40	103MB
Reddit	232,965	114,615,892	602	41	1.8GB

Table IV: A summary of the GCN model specifications.

Model	Layers	Hidden Dim.	Aggregation	Others/Details
GCN	2	16/64	Mean	16 for Cora/CiteS./Pub.; 64 for NELL/Reddit
GIN	3	16/64	Add	
GraphSAGE	2	16/64	Mean	
GAT	2	8	Attention	8 heads
ResGCN	28	128	Max	-

Baselines and Evaluation Metrics. To benchmark our GCoD with SOTA GCN acceleration works, we consider **a total of nine baselines**: PyTorch Geometric (PyG) [9] and Deep Graph Library (DGL) [37] on a Linux workstation with Intel Xeon E5-2680 v3 CPUs and NVIDIA RTX 8000 GPUs, respectively, and SOTA GCN accelerators HyGCN [42], AWB-GCN [13], and Deepburning-GL on three FPGA platforms (i.e., ZC706, KCU1500, and Alveo U50) [24]. The system configurations of the baselines and our GCoD are summarized in Tab. V. We evaluate all above platforms in terms of acceleration latency speedups, energy consumption, and required off-chip memory bandwidth and accesses. In addition, we compare the achieved accuracy of GCoD algorithm with SOTA GCN compression baselines, including RP [10], SGCN [23], QAT [8], and Degree-Quant [34].

Hardware Experiment Setup. To evaluate GCoD accelerator, we consider the standard FPGA evaluation and implementation flows in Vivado 2018.3 [40]. Specifically, we adopt a Xilinx VCU128 FPGA board [39], which is equipped with 9024 DSPs, 42MB on-chip memory, and 460 GB/s HBM off-chip memory, where the off-chip memory bandwidth is proportionally distributed across GCoD's sub-accelerators based on their assigned workloads/resources. For a fair comparison with AWB-GCN [13], GCoD accelerator is clocked at 330MHz and adopts 4096 PEs with a 32-bit fixed point precision. In addition, we discuss a GCoD variant that supports the quantized GCNs and termed as GCoD (8-bit). Quantization largely reduces the off-chip memory bandwidth requirement and thus enable GCoD (8-bit) to afford 10240 on-chip PEs ($\approx$ 5200 DSPs).

B. Overall Performance

We first evaluate GCoD against both the general platforms (PyG/DGL-CPU and PyG/DGL-GPU) and SOTA GCN accelerators in terms of speedup, off-chip memory bandwidth

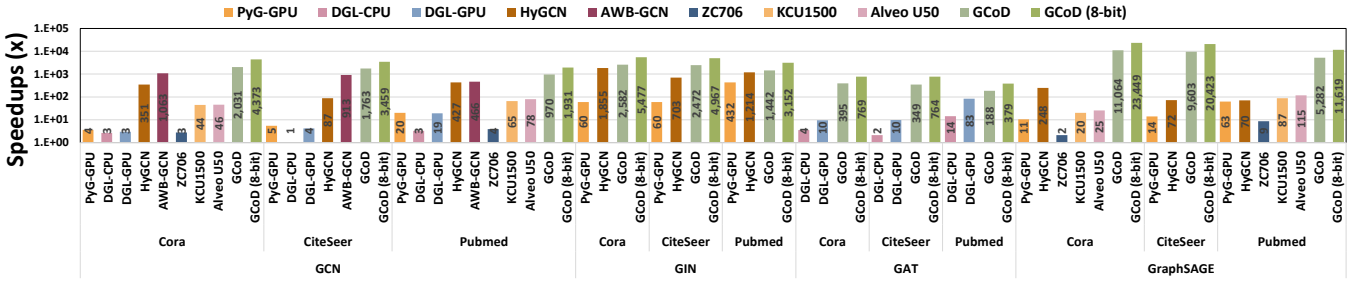

Figure 9: The normalized inference speedups (w.r.t. PyG-CPU) achieved by our GCoD framework over **nine** SOTA baselines on **four** variant GCN models and **three** representative citation graph datasets.

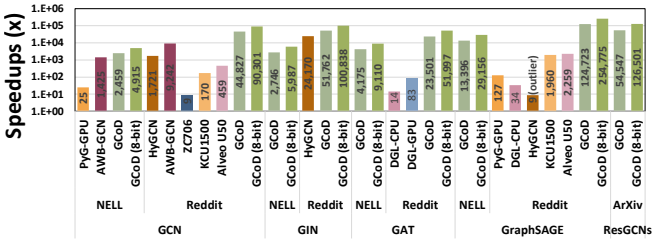

Figure 10: The normalized inference speedups (w.r.t. PyG-CPU) comparisons on **large** GCN models and graph datasets.

Table V: System configurations of the baselines and GCoD.

Design/Platform	Compute Unit	On-Chip Memory	Off-Chip Memory	Die Area (mm^2)	Power (W)
PyG/DGL-CPU	2.5GHz @24 cores	L1d/L1i: 24 x 32KB L2: 3MB L3: 30MB	1365.5 GB/s DDR4	-	150
PyG/DGL-GPU	1.35GHz @4352 cores	L1: 68 x 64KB L2: 5.5MB	616 GB/s GDDR6	754 (12nm)	250
HyGCN	1GHz@32 SIMD & 8 systolic array	Input: 128KB; Edge: 2MB; Weight: 2MB; Output: 4MB; Aggregation: 16MB	256 GB/s HBM~1.0	7.8 (12nm)	6.7
AWB-GCN	330MHz@Intel D5005 FPGA	4096 PEs 244 Mb Scratchpad	76.8 GB/s DDR4	-	215
ZC706	220MHz@900 DSPs	19.2MB	12.8GB/s DDR3	-	
KCU1500	5520 DSPs	75.9MB	76.8GB/s DDR4	-	
Alveo U50	5952 DSPs	227.3MB	316GB/s HBM~2.0	-	50
GCoD	330MHz@ VCU128	4096 PEs 9MB BRAM 33MB URAM	460GB/s HBM	-	180

*GCoD (8-bit) uses 10240 PEs as 8-bit saves required bandwidth.

requirement, and the number of off-chip memory accesses.

GCoD over CPU/GPU Platforms. Figs. 9 & 10 show the overall performance of our GCoD and the baselines. We can see that GCoD on-average achieves 15286×, 294×, 1057×, and 460× speedups over PyG-CPU, PyG-GPU, DGL-CPU, and DGL-GPU, respectively, while GCoD (8-bit) more aggressively achieves 32158×, 607×, 2213×, and 962× speedups over PyG-CPU, PyG-GPU, DGL-CPU, and DGL-GPU, respectively. The superior GCoD improvements validate the effectiveness of GCoD's dedicated algorithm and accelerator innovations: (1) GCoD's split and conquer training algorithm largely alleviates the irregularity of the graph adjacency matrices, leading to more consecutive address accesses of off-chip memory; and (2) GCoD accelerator enables more balanced workloads and higher utilization for each sub-accelerator, and allows more data reuses and lower on-chip storage demand.

Table VI: Speedup breakdown of GCoD accelerator w/ or w/o sparsification (SP.) and quantization (Quant.).

Methods	Speedups over PyG-CPU				
	Cora	CiteSeer	Pubmed	NELL	Reddit
AWB-GCN	1063×	913×	466×	1425×	9242×
GCoD Accele.	1824×	1692×	901×	2294×	39881×
GCoD Accele. w/ SP.	2031×	1763×	970×	2459×	44827×
GCoD Accele. w/ SP. & Quant.	4373×	3459×	1931×	4915×	90301×

GCoD over SOTA GCN Accelerators. We further compare GCoD with SOTA GCN accelerators: HyGCN [42], AWB-GCN [13], and Deepburning-GL on three FPGA platforms (ZC706, KCU1500, and Alveo U50) [24]. We follow most of these baselines to report the relative speedups over PyG-CPU on an Intel Xeon E5-2680 v3 CPU, and analyze the achieved improvements, as elaborated below.

(1) Speedup. As shown in Fig. 9 & 10, GCoD on-average achieves 7.8×, 2.5×, 2532×, 165×, and 115× speedups over HyGCN (without considering HyGCN's outlier when evaluated GraphSAGE), AWB-GCN, and Deepburning-GL on three FPGA platforms (ZC706, KCU1500, and Alveo U50), respectively. GCoD benefits are attributable to the dedicated algorithm and accelerator co-design. Specifically, HyGCN adopts coarse-grained block-wise scheduling while GCoD adopts fine-grained, adaptive row/column-wise pipeline; AWB-GCN realizes workload balance via on-the-fly auto-tuning while GCoD leverages a split and conquer algorithm to achieve naturally balanced workload; We further provide the improvement breakdown in Tab. VI, from which we can see that the improvement is mostly attributed to GCoD's two-pronged accelerator that leads to on-average 2.29× speedup over AWB-GCN [13], while sparsification further provides 1.09× speedups. Meanwhile, the GCoD (8-bit) variant offers an additional 2.02× on-average speedup.

(2) Memory Bandwidth Consumption and Accesses. Fig. 11 (a) shows the evaluation results in terms of off-chip memory bandwidth consumption. We can see that GCoD and GCoD (8-bit) only require on-average 48% and 26% off-chip memory bandwidth as compared to HyGCN, respectively.

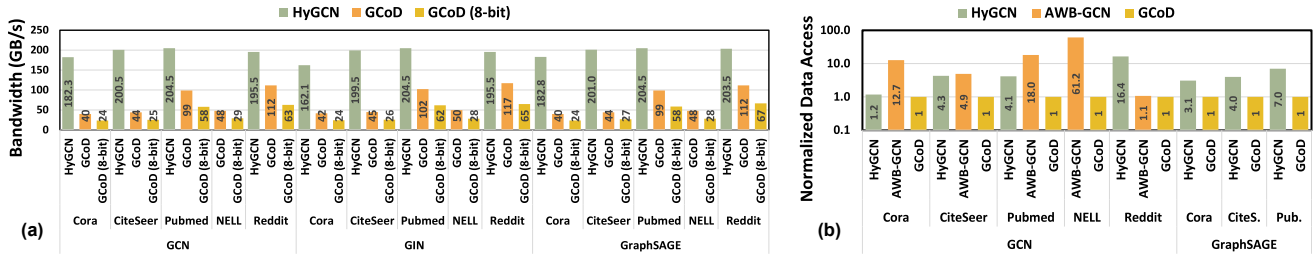

Figure 11: (a) Bandwidth requirement of GCoD and HyGCN and (b) normalized data access of GCoD, HyGCN, and AWB-GCN. Note that we record the peak bandwidth in (a), so the relative scale of (a) and (b) are slightly different.

Table VII: Comparison between GCoD with SOTA GCN compression methods, including Random Pruning (RP) [10], SGCN [23], QAT [8], and Degree-Quant [34].

Models	Methods	Accuracy (%)				
		Cora	CiteSeer	Pubmed	NELL	Reddit
GCN	Vanilla	81.1±1.2	70.2±0.8	79.1±0.6	65.6±0.7	92.2±1.1
	RP	79.6±0.8	70.4±0.5	78.4±0.6	63.5±2.3	91.2±2.2
	SGCN	80.2±0.7	70.4±0.7	79.1±0.1	64.2±1.2	91.3±1.3
	QAT	81.0±0.7	71.3±1.0	79.0±0.2	65.1±1.4	92.4±0.9
	Degree-Quant	81.7±0.7	71.0±0.9	79.1±0.1	65.2±0.8	92.6±1.5
	GCoD	**81.9±0.8**	**71.7±0.5**	79.5±0.3	**66.3±0.5**	**93.4±0.9**
	GCoD (8-bit)	81.0±0.9	70.6±0.3	**79.5±0.2**	66.0±0.3	93.2±1.3
	GCoD Improv.	**↑0.2% ~ ↑2.8%**				
GAT	Vanilla	83.1±0.4	72.2±0.7	78.8±0.3	66.6±0.3	94.2±0.3
	RP	80.9±0.6	69.8±0.8	78.2±0.1	64.5±1.2	93.1±1.2
	SGCN	81.9±0.3	71.9±0.2	78.4±0.1	64.9±1.0	93.4±0.9
	QAT	81.9±0.7	71.2±1.0	78.3±0.5	65.1±0.8	93.8±0.5
	Degree-Quant	82.7±0.7	71.6±1.0	78.6±0.3	65.9±0.6	94.0±0.7
	GCoD	**83.2±0.3**	**72.2±0.4**	**79.0±0.1**	**66.7±0.4**	94.5±0.2
	GCoD (8-bit)	82.6±0.2	71.8±0.1	78.8±0.2	66.5±0.2	**94.5±0.4**
	GCoD Improv.	**↑0.1% ~ ↑2.2%**				
GIN	Vanilla	78.6±0.9	67.5±1.5	78.5±0.2	65.2±0.2	92.8±2.2
	RP	74.6±0.4	64.5±0.5	76.9±0.6	64.2±0.5	92.0±0.5
	SGCN	78.0±0.1	67.0±0.1	77.2±1.1	64.8±0.4	92.3±0.9
	QAT	75.6±1.2	63.0±2.6	77.5±0.2	64.7±0.3	92.9±0.4
	Degree-Quant	78.7±1.4	67.5±1.4	78.1±0.5	65.2±0.3	93.1±0.6
	GCoD	**78.9±0.5**	68.6±0.8	**78.5±0.3**	**65.8±0.2**	93.3±0.9
	GCoD (8-bit)	78.4±0.2	**68.7±1.3**	78.3±0.2	65.6±0.4	**93.3±1.1**
	GCoD Improv.	**↑0.3% ~ ↑4.2%**				
GraphSAGE	Vanilla	81.2±0.2	71.1±0.3	78.7±0.2	66.2±0.4	93.8±2.9
	RP	77.7±0.7	66.1±2.2	76.0±0.3	65.5±0.4	90.7±1.8
	SGCN	79.2±0.5	70.9±0.3	78.5±0.1	66.1±0.3	93.9±0.9
	GCoD	**81.4±0.6**	71.3±0.3	**79.0±0.4**	**66.7±0.6**	**94.5±1.2**
	GCoD (8-bit)	80.8±0.4	**71.3±0.1**	78.8±0.1	66.4±0.8	94.3±1.5
	GCoD Improv.	**↑0.2% ~ ↑3.8%**				

The high bandwidth of HyGCN is attributed to the required high-degree parallelism, whereas GCoD accelerator's fine-grained pipelines enable more frequent data reuses, largely alleviating the off-chip bandwidth requirement. Fig. 11 (b) reports the measured off-chip memory accesses comparison for processing GCNs with GCoD, HyGCN, and AWB-GCN. Note that we count the number of off-chip accesses assuming that the input features and adjacency matrices are stored in the off-chip memory before processing. In practice, these matrices can be partially or entirely stored on-chip to reduce the data accesses and bandwidth requirements [13].

C. Evaluation of the GCoD Algorithm

GCoD over SOTA Compression Baselines. Table VII compares the accuracy of GCoD with SOTA compression methods to evaluate the effectiveness of GCoD algorithm. We can see that GCoD consistently achieves a comparable or even better accuracy (↑0.2% ~ ↑4.2%) over the vanilla GCNs and all compression baselines, while offering a 5% ~ 15% structural sparsity ratio (and thus more balanced workload).

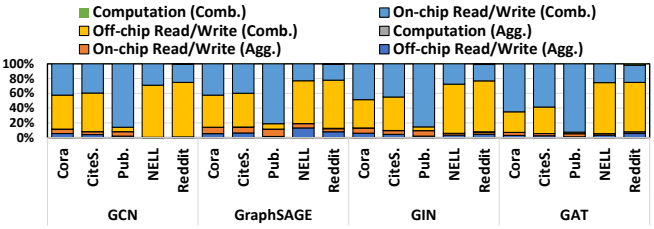

Figure 12: The energy breakdown of the GCoD framework when evaluated on the four GCN models and five graph datasets.

Ablation Studies of the Design Hyper-Parameters. Our GCoD algorithm has two hyper-parameters: the total number of classes C (i.e., sub-accelerators) and subgraphs S. To validate sensitivity of GCoD benefits, we measure the speedups and off-chip memory bandwidth requirements across a wide range of the design hyper-parameters $C \in \{1, 2, 3, 4\}$ and $S \in \{8, 12, 16, 20\}$, and find that GCoD consistently achieves 1.8× ~ 2.8× speedups over AWB-GCN and reduces the off-chip memory bandwidth by 26% ~ 53%, validating GCoD's general effectivenss and robustness.

D. Evaluation of the GCoD Accelerator

Energy Breakdown. Fig. 12 shows GCoD accelerator's energy breakdown in terms of computations and off-chip memory accesses in both the combination and aggregation phases. We can see that (1) the combination phase consumes most of the energy than the aggregation phase, thanks to GCoD acceleration (vs. PyG-CPU on which aggregation occupies 80% ~ 99% [42]), indicating the effectiveness of GCoD in alleviating the performance bottleneck due to the aggregation phase, and (2) the energy cost of accessing HBM remains reasonable as graph size increases, validating GCoD's scalability.

Resource-aware vs. Efficiency-aware Pipeline. As discussed in Sec. V-B, GCoD adopts efficiency-aware pipeline for small/medium graphs to achieve more data reuses at a cost of storing aggregation outputs on-chip. When processing large graphs, e.g., Reddit the outputs of which require

36MB storage and cannot be fully stored on-chip, GCoD uses resource-aware pipeline for better balancing the data reuses and on-chip storage requirement. The relatively more off-chip memory accesses when processing Reddit (see Fig. 11 (b)) is resulting from the less data resues for reduced on-chip storage.

VII. Conclusion

We propose, develop, and validate GCoD, an algorithm and accelerator co-design framework. On the algorithm level, GCoD integrates a split and conquer GCN training strategy to polarize the graphs to be either denser or sparser in local neighborhoods without compromising the model accuracy, resulting in graph adjacency matrices that have merely two levels of balanced workload and thus enjoy largely enhanced regularity. On the hardware level, GCoD integrates a dedicated two-pronged accelerator with a dedicated engine to process each of the aforementioned workloads, further boosting the overall utilization and acceleration efficiency. Extensive experiments and ablation studies validate the advantages of GCoD.

Acknowledgment

We would like to acknowledge the funding support from the NSF RTML program (Award number: 1937592) and the NSF NeTS program (Award number: 1801865) for this project. The authors also thank our colleague Mr. Cheng Wan at Rice University for his help and discussion in the graph reordering algorithm.

References

[1] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 22–31.

[2] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020.

[3] A. A. Awan, H. Subramoni, and D. K. Panda, "An in-depth performance characterization of cpu- and gpu-based dnn training on modern architectures," in *Proceedings of the Machine Learning on HPC Environments*, ser. MLHPC'17. Association for Computing Machinery, 2017.

[4] A. Azad, M. Jacquelin, A. Buluç, and E. G. Ng, "The reverse cuthill-mckee algorithm in distributed-memory," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017.

[5] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.

[6] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. Hruschka, and T. Mitchell, "Toward an architecture for never-ending language learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, no. 1, 2010.

[7] X. Chen, Y. Wang, X. Xie, X. Hu, A. Basak, L. Liang, M. Yan, L. Deng, Y. Ding, Z. Du *et al.*, "Rubik: A hierarchical architecture for efficient graph learning," *arXiv preprint arXiv:2009.12495*, 2020.

[8] A. Fan, P. Stock, B. Graham, E. Grave, R. Gribonval, H. Jegou, and A. Joulin, "Training with quantization noise for extreme model compression," in *ICLR*, 2021.

[9] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[10] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=rJl-b3RcF7

[11] Y. Gao, H. Yang, P. Zhang, C. Zhou, and Y. Hu, "Graphnas: Graph neural architecture search with reinforcement learning," *arXiv preprint arXiv:1904.09981*, 2019.

[12] R. Garg, E. Qin, F. M. Martínez, R. Guirado, A. Jain, S. Abadal, J. L. Abellán, M. E. Acacio, E. Alarcón, S. Rajamanickam *et al.*, "A taxonomy for classification and comparison of dataflows for gnn accelerators," *arXiv preprint arXiv:2103.07977*, 2021.

[13] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt *et al.*, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *53rd IEEE/ACM Int. Symp. Microarchit.(MICRO)*, 2020, pp. 1–15.

[14] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[15] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[16] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.

[17] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[18] K. Kersting, N. M. Kriege, C. Morris, P. Mutzel, and M. Neumann, "Benchmark data sets for graph kernels," 2020, http://graphkernels.cs.tu-dortmund.de.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[20] K. Kiningham, C. Re, and P. Levis, "Grip: A graph neural network acceleratorarchitecture," *arXiv preprint arXiv:2007.13828*, 2020.

[21] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[22] G. Li, C. Xiong, A. Thabet, and B. Ghanem, "Deepergcn: All you need to train deeper gcns," *arXiv preprint arXiv:2006.07739*, 2020.

[23] J. Li, T. Zhang, H. Tian, S. Jin, M. Fardad, and R. Zafarani, "Sgcn: A graph sparsifier based on graph convolutional networks," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2020, pp. 275–287.

[24] S. Liang, C. Liu, Y. Wang, H. Li, and X. Li, "Deepburning-gl: an automated framework for generating graph neural network accelerators," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

[25] S. Liang, Y. Wang, C. Liu, L. He, L. Huawei, D. Xu, and X. Li, "Engn: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Transactions on Computers*, 2020.

[26] Y. Lin, C. Sakr, Y. Kim, and N. Shanbhag, "Predictivenet: An energy-efficient convolutional neural network via zero prediction," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[27] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 389–400. [Online]. Available: https://doi.org/10.1145/3210240.3210337

[28] Y. Liu, N. Zhang, D. Wu, A. Botterud, R. Yao, and C. Kang, "Guiding cascading failure search with interpretable graph convolutional network," *arXiv preprint arXiv:2001.11553*, 2020.

[29] W. Peng, X. Hong, H. Chen, and G. Zhao, "Learning graph convolutional network for skeleton-based human action recognition by neural searching." in *AAAI*, 2020, pp. 2669–2676.

[30] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI magazine*, vol. 29, no. 3, pp. 93–93, 2008.

[31] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 535–547.

[32] M. Simonovsky and N. Komodakis, "Dynamic edge-conditioned filters in convolutional neural networks on graphs," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[33] S. A. Tailor, J. Fernandez-Marques, and N. D. Lane, "Degree-quant: Quantization-aware training for graph neural networks," *arXiv preprint arXiv:2008.05000*, 2020.

[34] ——, "Degree-quant: Quantization-aware training for graph neural networks," in *ICLR*, 2021. [Online]. Available: https://openreview.net/forum?id=NSBrFgJAHg

[35] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[36] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[37] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[38] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[39] Xilinx Inc., "Virtex ultrascale+ hbm vcu128 fpga evaluation kit," https://www.xilinx.com/products/boards-and-kits/vcu128.html, (Accessed on 09/30/2020).

[40] ——, "Vivado Design Suite - HLx Editions," https://www.xilinx.com/products/design-tools/vivado.html, accessed 2019-09-16.

[41] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=ryGs6iA5Km

[42] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 15–29.

[43] H. Yang, "Aligraph: A comprehensive graph neural network platform," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 3165–3166.

[44] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," in *Advances in neural information processing systems*, 2018.

[45] H. You, C. Li, P. Xu, Y. Fu, Y. Wang, X. Chen, R. G. Baraniuk, Z. Wang, and Y. Lin, "Drawing early-bird tickets: Toward more efficient training of deep networks," in *ICLR*, 2020. [Online]. Available: https://openreview.net/forum?id=BJxsrgStvr

[46] H. You, Z. Lu, Z. Zhou, and Y. Lin, "GEBT: Drawing early-bird tickets in graph convolutional network training," *arXiv preprint arXiv:2103.00794*.

[47] H. Zeng and V. Prasanna, "Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 255–265.

[48] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Accurate, efficient and scalable graph embedding," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019.

[49] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[50] Y. Zhang, H. You, Y. Fu, T. Geng, A. Li, and Y. Lin, "G-CoS: Gnn-accelerator co-search towards both better accuracy and efficiency," *CoRR*, vol. abs/2109.08983, 2021. [Online]. Available: https://arxiv.org/abs/2109.08983

[51] C. Zheng, B. Zong, W. Cheng, D. Song, J. Ni, W. Yu, H. Chen, and W. Wang, "Robust graph representation learning via neural sparsification."