Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)
Survey Track

Survey on Graph Neural Network Acceleration: An Algorithmic Perspective

Xin Liu[1,2], **Mingyu Yan**[1*], **Lei Deng**[3], **Guoqi Li**[2,4], **Xiaochun Ye**[1,2], **Dongrui Fan**[1,2], **Shirui Pan**[5] and **Yuan Xie**[6]
[1] SKLCA, Institute of Computing Technology, Chinese Academy of Sciences, China
[2] University of Chinese Academy of Sciences, China
[3] Tsinghua University, China
[4] Institute of Automation, Chinese Academy of Sciences, China
[5] Monash University, Australia
[6] University of California, Santa Barbara, America
{liuxin19g, yanmingyu, yexiaochun, fandr}@ict.ac.cn, leideng@mail.tsinghua.edu.cn, guoqi.li@ia.ac.cn, shirui.pan@monash.edu, yuanxie@ucsb.edu

Abstract

Graph neural networks (GNNs) have been a hot spot of recent research and are widely utilized in diverse applications. However, with the use of huger data and deeper models, an urgent demand is unsurprisingly made to accelerate GNNs for more efficient execution. In this paper, we provide a comprehensive survey on acceleration methods for GNNs from an algorithmic perspective. We first present a new taxonomy to classify existing acceleration methods into five categories. Based on the classification, we systematically discuss these methods and highlight their correlations. Next, we provide comparisons from aspects of the efficiency and characteristics of these methods. Finally, we suggest some promising prospects for future research.

1 Introduction

Graph Neural Networks (GNNs) [Scarselli *et al.*, 2008] are deep learning based models that apply neural networks to graph learning and representation. They are technically skillful [Kipf and Welling, 2017; Veličković *et al.*, 2018] and theoretically supported [Pope *et al.*, 2019; Ying *et al.*, 2019], holding state-of-the-art performance on diverse graph-related tasks [Hamilton *et al.*, 2017; Xu *et al.*, 2019]. Owing to the significant success of GNNs in various applications, recent years have witnessed increasing research interests in GNNs, hastening the emergence of reviews that focus on different research areas. A few reviews [Wu *et al.*, 2020; Zhang *et al.*, 2020; Battaglia *et al.*, 2018] pay close attention to GNN models and generic applications, while others [Wang *et al.*, 2021b; Zhang *et al.*, 2021] place emphasis on specific usages of GNNs. Moreover, hardware-related architectures [Abadal *et al.*, 2021; Han *et al.*, 2021] and software-related algorithms [Lamb *et al.*, 2020] of GNNs are also emphatically surveyed by researchers. Thereby, the above reviews further promote the widespread use of GNNs. However, as GNNs are widely used in emerging scenarios, it is discovered that GNNs are plagued by some obstacles that lead to slow execution. Next, we will discuss obstacles that restrict the efficiency of GNNs execution and present our motivation.

Motivation: why GNNs need acceleration?

First, **explosive increase of graph data** poses a great challenge to GNN training on large-scale datasets. Previously, many graph-based tasks were often conducted on toy datasets that are relatively small compared to graphs in realistic applications, which is harmful to model scalability and practical usages. Currently, large-scale graph datasets are thereby proposed in literature [Hu *et al.*, 2020a] for advanced research, and at the same time, making GNNs execution (i.e., training and inference) a time-consuming process. *Second*, under the condition that the over-smoothing issue has been skillfully avoided [Rong *et al.*, 2020], using **deeper and more complicated structures** is a promising way to acquire a GNN model with good ability of expression [Chiang *et al.*, 2019; Rong *et al.*, 2020], which, on the other hand, will increase the time cost of training a well-expressive model. *Third*, special devices, such as edge devices, generally have **strict time restrictions on GNN training and inference**, especially in a time-sensitive task. Due to the limited computing and storage resources, the training and inference time on such devices can easily become intolerable. Therefore, it is still an urgent need to accelerate GNNs in both training and inference.

However, no literature has systematically investigated acceleration methods for GNNs at the algorithm level. Practically, algorithm level optimizations not only promote the model accuracy but also accelerate the model learning [Chen *et al.*, 2018b; Bojchevski *et al.*, 2020]. We argue that algorithmic acceleration methods for GNNs will greatly benefit processes of training and inference, and at the same time, the overall performance of GNN frameworks [Fey and Lenssen, 2019; Wang *et al.*, 2019a], since a well-designed framework equipped with an optimized algorithm can empirically gain a two-fold promotion [Lin *et al.*, 2020]. In consequence, despite some insightful reviews on graph-related frameworks and hardware accelerators, a thorough review on algorithmic acceleration methods for GNNs is highly expected, which is

*Corresponding author

Notations	Descriptions
$\mathbf{H}$	Hidden feature matrix of graph
$\mathbf{h}$	Hidden feature of a node
$\mathbf{X}$	Feature matrix of graph
$\mathbf{A}$	Original adjacency matrix
$\mathbf{A}_{sp}$, $\widetilde{\mathbf{A}}$	Sparsified and normalized adjacency matrix
$\mathbf{D}$, $\widetilde{\mathbf{D}}$	Degree matrix of $\mathbf{A}$ and $\widetilde{\mathbf{A}}$
$\mathbf{S}$	Normalized adjacency matrix with self-loops
$\mathbf{W}$	Weight matrix of graph
σ	Nonlinear activation function
V, E	Node and edge sets of a graph
$N(v)$, $SN(v)$	Original and sampled sets of v's neighbors

Table 1: Notations and corresponding descriptions used in the paper.

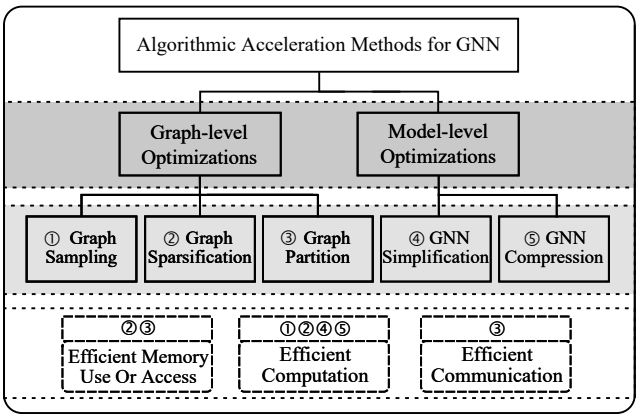

Figure 1: Taxonomy of acceleration methods for GNNs. These methods are classified into five categories by a double-level decision, i.e., the optimized factor in GNN execution (1st level) and the core mechanisms of these methods (2nd level). Moreover, five categories of methods are highlighted by their objectives. For instance, methods of "Graph Sparsification" and "Graph Partition" can speed up GNN execution by adopting efficient memory access or usage.

exactly the goal and the focus of this work.

In this paper, we provide a comprehensive survey on algorithmic acceleration methods for GNNs, in which graph-level and model-level optimizations are emphatically focused. To summarize, we highlight our contributions as follows:

1) **New Taxonomy:** we classify existing methods into five categories via a double-level taxonomy that jointly considers optimized factors and core mechanisms (see Section 2).

2) **Comprehensive Review:** we provide a comprehensive survey on existing methods and introduce these methods by categories. And we emphatically focus on common grounds and unique points among these methods (see Section 3).

3) **Thorough Comparison:** we summarize the performance of training time of typical acceleration methods and further give a thorough comparison from an overall perspective, in which correlations among these methods are particularly highlighted (see Section 4).

4) **Future Prospects:** based on the overall comparison, we discuss some potential prospects of GNNs acceleration for reference (see Section 5).

2 Preliminary and Taxonomy

In this section, we first introduce the background of GNNs and the conventional execution including processes of training and inference. Then, we propose a taxonomy of acceleration methods for GNNs. Notations and the corresponding descriptions used in the rest of the paper are given in Table 1.

2.1 Background of GNNs and Model Execution

To efficiently capture hidden patterns in graphs, GNNs provide an inspired idea of combining the design of modern neural networks and graph learning [Wu *et al.*, 2020]. With the background of deep learning, many variants of GNN are proposed by adding particular mechanisms to the original GNN model, e.g., Graph Convolutional Networks (GCNs) [Kipf and Welling, 2017], Graph Attention Networks (GATs) [Veličković *et al.*, 2018], and Graph Isomorphism Networks (GINs) [Xu *et al.*, 2019]. As with most artificial neural networks (ANNs), the execution of GNNs contains processes of training and inference. Herein, we take the GCN model as an exemplar for formulated introduction, owing to its powerful ability and the widespread usage of handling graph-related tasks. Generally, given $\mathbf{A}$ and $\mathbf{X}$ as input, the forward propagation in the l-th layer in GCN training can be formulated as:

$$\mathbf{H}^l = \sigma\left(\widetilde{\mathbf{D}}^{-1/2}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-1/2}\mathbf{H}^{l-1}\mathbf{W}^{l-1}\right). \quad (1)$$

The backpropagation is performed by updating $\mathbf{W}$ via the computed gradient. Distinctly, Equation 1 implies the forward propagation to be an iterative computing process in a layered manner, which takes non-trivial cost in terms of time and storage. As for inference, the trained model is utilized to infer and acquire hidden representation for downstream tasks. In consequence, for training, it is time-consuming to obtain the well-trained $\mathbf{W}$; for inference, attempts of deploying the inference to resource-limited devices make an urgent demand for inference efficiency.

2.2 Taxonomy of Acceleration Methods

A double-level taxonomy of existing algorithmic acceleration methods for GNNs is illustrated in Figure 1. First, methods are classified into two major categories according to the optimized factor. "Graph-level" denotes that the optimization is conducted on graphs used for training and inference by modifying topology or density of graphs. "Model-level" denotes that the optimization is made on GNNs' model containing modifications to the model structure or weight. Further, we divide these methods into five categories based on their mechanisms, i.e., graph sampling, graph sparsification, graph partition, GNN simplification, and GNN compression. As an exemplar, "Graph Sampling" denotes that graph sampling methods are utilized to accelerate the training convergence of GNNs. At the final level, we label these methods by their optimization objectives, e.g., graph sampling gains acceleration by reducing the computation cost. Detailed discussions are given in Section 3.

3 Acceleration Methods for GNNs

In this section, we discuss five types of algorithmic acceleration methods that respectively focus on graph-level and model-level optimizations. For each category of methods, we first introduce the fundamental and the way for acceleration from an overall aspect. Then, we exemplify typical work belonging to these categories and highlight their correlations.

3.1 Graph-level Optimizations

Graph Sampling

Conventional training of GNNs, especially GCNs, is executed in a full-batch manner, restricting model update to once per epoch and thus slows down the training convergence. As illustrated in Figure 2(a), graph sampling methods generally select partial nodes in one node's neighborhood or one layer in a graph to acquire subgraphs for subsequent training, which makes **efficient computation** for model training. We formulate the sampling-based training process of GNNs using GraphSAGE [Hamilton *et al.*, 2017] as an exemplar:

$$SN\left(v\right) = \texttt{Sampling}^{(l)}\left(N\left(v\right)\right) \tag{2}$$

$$\mathbf{h}_{N(v)}^{(l)} = \texttt{Aggregate}^{(l)}(\{\mathbf{h}_u^{(l-1)} : u \in SN(v)\}) \tag{3}$$

$$\mathbf{h}_v^{(l)} = \texttt{Update}^{(l)}(\mathbf{h}_v^{(l-1)}, \mathbf{h}_{N(v)}^{(l)}) \tag{4}$$

Functions `Aggregate` and `Update` denote processes for aggregating hidden features from neighbors and updating features, respectively. As we can observe, the sampling method restricts size of $N(v)$ to $SN(v)$ and decreases the computation cost in subsequent step, instead of generating $\mathbf{h}$ conventionally with a whole graph utilized. Moreover, sampling methods utilize a mini-batch strategy in training and update the model once per batch, which accelerates the model convergence and promotes the training efficiency.

Please note that, sampling methods for GNN training are generally diverse and are different in design purposes, e.g., for accelerating model training or reducing memory usage. Herein, we pay close attention to methods that benefit the training speed and convergence. Next up, by referring to the practice of previous literature [Liu *et al.*, 2021b], we orderly discuss these methods as follows.

• ***Node-wise sampling methods:*** node-wise sampling is a fundamental sampling method that focuses on each node and its neighbors in a training graph. GraphSAGE [Hamilton *et al.*, 2017] is an inductive learning framework in which a sampling method and a mini-batch strategy are first proposed to benefit the training. In the sampling process, GraphSAGE randomly selects 2-hop neighbors for each node in a batched manner. The aggregation process is performed based on the sampled result. Taking inspiration from GraphSAGE, VR-GCN [Chen *et al.*, 2018a] restricts the number of neighbors per node to an arbitrarily small size for alleviating the exponential growth of the receptive field.

• ***Layer-wise sampling methods:*** layer-wise sampling can be regarded as an algorithmic improvement of node-wise sampling. Since node-wise sampling suffers from exponential expansion of multi-hop neighbors, layer-wise sampling alleviates the heavy overhead by sampling a fixed number of nodes in each layer based on pre-computed probability. Typically, FastGCN [Chen *et al.*, 2018b] independently samples a certain number of nodes per layer and reconstructs connections (of nodes) between two successive layers according to $\mathbf{A}$ of the training graph. Based on a hierarchical model as well, the sampling process of AS-GCN [Huang *et al.*, 2018] is layer-dependent and probability-based. It samples nodes according to the parent nodes sampled in the upper layer.

• ***Subgraph-based sampling methods:*** subgraph-based sampling generates subgraphs for training in a two-step manner: sampling nodes and constructing connections (edges). The graph samplers are varied compared to the first two sampling methods. For instance, GraphSAINT [Zeng *et al.*, 2020] utilizes three samplers, i.e., node sampler, edge sampler, and random walk sampler, to sample nodes (or edges) and construct a subgraph in each batch. Moreover, subgraph sampling can be parallelized at the processing unit level with a training scheduler aided [Zeng *et al.*, 2019], making the training efficient and easily scalable. Such parallelization takes good advantage of the property that subgraphs can be sampled independently.

• ***Heterogeneous sampling methods:*** heterogeneous sampling is designed to accelerate the training and handle the graph heterogeneity, i.e., imbalanced number and type of neighbors in a node's neighborhood. HetGNN [Zhang *et al.*, 2019] resolves the heterogeneity issue by traversing and collecting neighbors in a random walk manner. The collected neighbors are grouped by type and are further sampled based on the visit frequency. To achieve balanced neighboring distribution, HGSampling is proposed in HGT [Hu *et al.*, 2020b] to sample different types of nodes orderly. HGSampling is probability-based and ensures a balanced sampling result among diverse neighbors.

Graph Sparsification

Graph sparsification is a classic technique for speeding up many fully dynamic graph algorithms [Eppstein *et al.*, 1997]. As illustrated in Figure 2(b), graph sparsification methods typically remove task-irrelevant edges in a graph by designing a specific optimization goal. Recent graph sparsification methods propose to sparsify input graphs before they are fed into a GNN model, which makes **efficient computation and memory access** for model training. To be generic, we formulate the sparsification method as follows:

$$\mathbf{A}_{sp} = \texttt{Sparse}(\mathbf{A}) = \begin{cases} \texttt{Sp.Algo.}(\mathbf{A}), & \textit{Heuristic} \\ \texttt{Sparsifier}(\mathbf{A}), & \textit{Learnable} \end{cases} \tag{5}$$

$$\{\mathbf{A}_{sp} \rightarrow GNN\} \rightarrow \textit{Training \& Inference} \tag{6}$$

In Equation 5, the `Sparse` function can be implemented via two schemes: 1) designing a heuristic sparsification algorithm; 2) building a learnable module (e.g., sparsifier) for sparsification. After processing, the graph ($\mathbf{A}_{sp}$) is highly sparse in general, where task-irrelevant edges are removed to reduce subsequent computation and redundant memory access cost in GNN. Moreover, graph sparsification methods can be skillfully leveraged to reduce the communication latency during training, thus accelerating GNN training in the hardware [Arka *et al.*, 2021]. Next, we discuss these methods according to their schemes (i.e., heuristic or learnable).

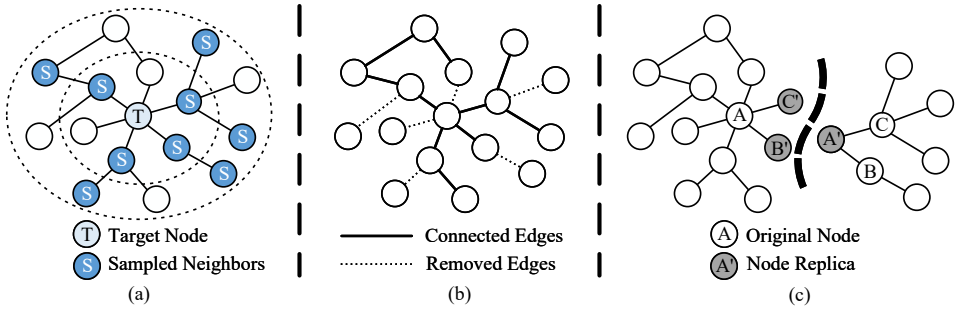

Figure 2: Illustrations of graph-level improvements: (a) a graph sampling method that samples 2-hop neighbors; (b) a graph sparsification method that removes useless edges; (c) a graph partition method that divides a graph into two subgraphs with node replicas preserved.

• ***Heuristic sparsification:*** DropEdge [Rong *et al.*, 2020] proposes to randomly remove edges in each training epoch, which aims to resolve the over-smoothing issue in deep GNN training. Instead of designing a global optimization goal, random edge dropping is heuristic and ensures both fast execution of algorithm and randomness of graph. FastGAT [Srinivasa *et al.*, 2020] presents a resistance-based spectral graph sparsification solution to remove useless edges, reducing the number of attention coefficients in a GAT model. Thereby, both training and inference are well accelerated.

• ***Learnable module for sparsification:*** NeuralSparse [Zheng *et al.*, 2020a] and SGCN [Li *et al.*, 2020] cast graph sparsification as an optimization problem. NeuralSparse utilizes a deep neural network (DNN) to learn a sparsification strategy based on the feedback of downstream tasks in training. It casts graph sparsification as an approximative objective of generating sparse k-neighbor subgraphs. SGCN formulates graph sparsification as an optimization problem and resolves it via an alternating direction method of multipliers (ADMM) approach [Boyd *et al.*, 2011]. Additionally, GAUG [Zhao *et al.*, 2021] proposes two variants: GAUG-M utilizes an edge predictor to acquire probabilities of edges in a graph, and modifies input graphs based on the predicted probabilities; GAUG-O integrates the edge predictor and GNN model to jointly promote edge prediction and model accuracy.

Other literature aims to accelerate GNN inference, such as UGS [Chen *et al.*, 2021] and AdaptiveGCN [Li *et al.*, 2021]. UGS utilizes lottery ticket hypothesis to sparsify input graphs and a GNN model iteratively. AdaptiveGCN builds an edge predictor module to remove task-irrelevant edges for inference acceleration on both CPU and GPU platforms.

Graph Partition

Graph partition has been an NP-hard problem with the goal of reducing the original graph to smaller subgraphs that are constructed by mutually exclusive node groups [Buluç *et al.*, 2016]. As graph data goes enormous, studies have previously deployed GNN execution on diverse platforms, such as a distributed system with multiple GPUs equipped, which places high demands for data communication in systems. Thereby, to accelerate GNN execution on such systems, graph partition methods are introduced to reduce the communication cost and maintain the load balance (**efficient communication and memory usage**). Figure 2(c) illustrates a simple paradigm of graph partition using a vertex-cut strategy. We generally formulate the process of graph partition as follows:

$$V_1 \cup \cdots \cup V_k = \texttt{Divide}(V), \{V_i \cap V_j = \emptyset \quad \forall i \neq j\} \quad (7)$$

$$\{Block_i | i \geq 1\} = \texttt{LoadBalance}(V_1, \cdots, V_k) \quad (8)$$

$$\{Subgraph_i | i \geq 1\} = \texttt{Partition}(E, \{Block_i | i \geq 1\}) \quad (9)$$

The $Block_i$ here is a set of divided nodes used to generate a $Subgraph_i$. The generated sugraphs can be deployed on different devices for distributed training. Since graph partition methods are generally diverse and are widely used in a distributed system, one can easily find some classic partition methods, such as METIS [Karypis and Kumar, 1998], are well employed to reduce the communication cost. Herein, we discuss typical partition methods for efficient GNN training that focus on different optimization targets.

• ***Graph partition for generic GNN training:*** DistGNN [Md *et al.*, 2021] applies a vertex-cut graph partition method to full-batch GNN training to reduce communication across partitions. The partition method preserves replicas for nodes segmented with their neighbors, which benefits the execution of delayed remote partial feature aggregation. DistDGL [Zheng *et al.*, 2020b] partitions a graph using METIS, and further colocates nodes/edges features with graph partitions. To reduce the number of cross-partition edges and achieve a good balance, DistDGL formulates the objective as a multi-constraint problem. In this way, the original METIS is improved to solve the edge balance issue in graph partition during training.

• ***Graph partition for sampling-based GNN training:*** sampling is becoming a time-consuming process in training large-scale graphs. BGL [Liu *et al.*, 2021a] utilizes a graph partition method to minimize cross-partition communication in sampling subgraphs during GNN training, in which multi-source breadth first search (BFS) and greedy assignment heuristics are fully leveraged to ensure both multi-hop locality and balanced nodes distribution. Cluster-GCN [Chiang *et al.*, 2019] partitions a graph into multiple clusters using METIS, after which these clusters are randomly sampled to construct subgraphs for subsequent training. Cluster-GCN reduces memory usage in training and achieves a fast training speed on a deep GCN model. Pagraph [Lin *et al.*, 2020] proposes a GNN-aware graph partition method to ensure balanced partitions in workload and avoid cross-partition visits in the sampling process as much as possible. Specifically,

partitioned subgraphs are extended to include L-hop neighbors and corresponding edges for an L-layer model, allowing independent training on each graph partition.

3.2 Model-level Optimizations

GNN Simplification

GNN simplification is a model-specific method that simplifies operation flows in an GNN, targeting the improvement of **computation efficiency** in GNN training and inference. The layer propagation in a widely used GNN model, i.e., GCN, is given in Equation 1, in which linear aggregation of neighboring information in spatial dimension and nonlinear activation are combined to update node representations. However, through recalling the design of classic yet straightforward classifiers, recent literature has argued that the efficiency of GNNs can be further promoted by simplifying redundant operations, despite the current outstanding performance. Since simplified GNNs are used in different tasks, we discuss typical models according to the model generality (or specificity).

• ***Generic simplified GNN:*** SGC [Wu *et al.*, 2019] removes the nonlinear activation, i.e., ReLU, between each layer to decrease model complexity, with only the final `Softmax` function preserved to generate probabilistic outputs. The simplified propagation is given as follows:

$$\mathit{Output} = \texttt{Softmax}(\mathbf{S}\cdots\mathbf{SSXW}^1\mathbf{W}^2\cdots\mathbf{W}^l) \qquad (10)$$

The linearized model is lightweight and includes a parameter-free part, where the initial $\mathbf{X}$ times $\mathbf{S}$ can be pre-processed without using weight matrices $\mathbf{W}$. Leveraging above properties, SGC achieves significant acceleration on the training speed while maintaining comparable accuracy in many generic tasks, such as text and graph classification.

• ***Special simplified GNN with tasks related:*** LightGCN [He *et al.*, 2020] and UltraGCN [Mao *et al.*, 2021] aim to simplify the GNN model for learning embeddings from user-item interaction graphs in a recommender system. In a generic GCN layer, aggregated features are further processed via two operations: linear transformation using a learned $\mathbf{W}$ and nonlinear activation. LightGCN finds that feature transformation and nonlinear activation hardly benefit collaborative filtering by empirically exploring ablation studies on NGCF [Wang *et al.*, 2019b]. Therefore, LightGCN abandons the above two operations and drops self-loops in $\mathbf{A}$ to simplify the GCN model used in collaborative filtering. Moreover, UltraGCN identifies the issue in message passing of LightGCN and resolves it with a simpler structure. LightGCN passes messages by stacking multiple layers, which can cause an over-smoothing problem and is harmful to training efficiency. UltraGCN thus proposes to abandon explicit message passing among multiple layers and get approximate ultimate embeddings directly, yielding 14× speedup over LightGCN in terms of training.

GNN Compression

Current deep learning applications heavily rely on enormous data and complicated models in general. Such a model is well-representative but contains hundreds of millions of parameters, making the model training an intolerable time-consuming process. Model compression is a technique that compresses a complicated model, such as a DNN [Cheng *et al.*, 2017], to a lightweight one with typically fewer parameters preserved, which is widely used to yield acceleration in training and inference and save the computation cost. By reviewing an emerging trend of applying model compression to GNNs for **efficient computation**, we discuss these methods according to their mechanisms.

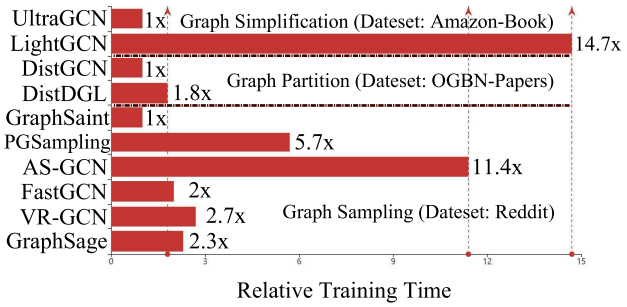

Figure 3: Comparison of training time among typical methods.

• ***Model quantification:*** as a particular quantification technique, binarization skillfully compresses model parameters and graph features in GNNs to yield significant acceleration in inference. Binarized DGCNN [Bahri *et al.*, 2021] and Bi-GCN [Wang *et al.*, 2021a] similarly introduce binarization strategies into GNNs to speed up model execution and reduce memory consumption. Degree-quant [Tailor *et al.*, 2021] proposes a quantization-aware training method for GNNs to enable model inference with low precision integer (INT8) arithmetic, achieving up to 4.7× speedup on CPU platform.

• ***Knowledge distillation:*** knowledge distillation (KD) [Hinton *et al.*, 2015] is a technique that extracts knowledge from a teacher model and injects it into a smaller one with similar performance maintained, which at the same time, yields acceleration on model inference. Yang et al. [Yang *et al.*, 2020] propose to preserve the local structure of a teacher model via a special-designed module, which helps the knowledge transfer from a trained larger model to a smaller one. Other literature [Yang *et al.*, 2021] presents an effective KD framework where a specially built student model can jointly benefit from a teacher model and prior knowledge. Moreover, TinyGNN [Yan *et al.*, 2020a] bridges the gap of extracting neighbor information between a teacher mode and a student model via a combined use of a peer aware module and a neighbor distillation strategy. The learned student model can achieve dozens of times speedup in inference than the teacher model.

4 Comparison and Analysis

In this section, comparison of model efficiency (reflected by relative training time) is given in Figure 3. Note that methods in different categories are partitioned by chain lines and these in the same category are compared using the same dataset and platform. All data is collected from literature [Liu *et al.*, 2021b; Mao *et al.*, 2021; Md *et al.*, 2021]. We also provide overall summary and comparison of existing methods in Tables 2 & 3. We pay special attention to the following aspects.

• ***GNN backbone*** denotes which models can be applied with acceleration methods. *Sampling* methods are generally con-

Method	Work
Graph Sampling	GraphSAGE[1], VR-GCN[2], FastGCN[3], AS-GCN[4], PGSampling[5], GraphSAINT[6], HetGNN[7], HGT[8]
Graph Sparsification	DropEdge[9], FastGAT[10], NeuralSparse[11], SGCN[12], GAUG[13], UGS[14], AdaptiveGCN[15]
Graph Partition	DistGNN[16], DistDGL[17], BGL[18], Cluster-GCN[19], Pagraph[20]
GNN simplification	SGC[21], LightGCN[22], UltraGCN[23]
GNN compression	Binarized DGCNN[24], Bi-GCN[25], Degree-quant[26], KD-GCN[27], KD-framework[28], TinyGCN[29]

1:[Hamilton *et al.*, 2017],2:[Chen *et al.*, 2018a],3:[Chen *et al.*, 2018b],4:[Huang *et al.*, 2018],5:[Zeng *et al.*, 2019],6:[Zeng *et al.*, 2020], 7:[Zhang *et al.*, 2019],8:[Hu *et al.*, 2020b],9:[Rong *et al.*, 2020],10:[Srinivasa *et al.*, 2020],11:[Zheng *et al.*, 2020a],12:[Li *et al.*, 2020], 13:[Zhao *et al.*, 2021],14:[Chen *et al.*, 2021],15:[Zheng *et al.*, 2020a],16:[Md *et al.*, 2021],17:[Zheng *et al.*, 2020b],18:[Liu *et al.*, 2021a], 19:[Chiang *et al.*, 2019],20:[Lin *et al.*, 2020],21:[Wu *et al.*, 2019],22:[He *et al.*, 2020],23:[Mao *et al.*, 2021],24:[Bahri *et al.*, 2021], 25:[Wang *et al.*, 2021a],26:[Tailor *et al.*, 2021],27:[Yang *et al.*, 2020],28:[Yang *et al.*, 2021],29:[Yan *et al.*, 2020a]

Table 2: Summary and classification of the acceleration methods and corresponding work.

Method	GNN Backbone	Acceleration Phase	Optimization Obj.	General App.	Special App.
Graph Sampling	GCN, GAT	Train.	Compt.	Node Classification	Variance Elimination
Graph Sparsification	GCN, GAT, GIN	Train. & Infer.	Mem. & Compt.	Node Classification	Denoising
Graph Partition	GCN, GAT	Train.	Mem. & Com.	Node Classification	Clustering
GNN Simplification	GCN	Train. & Infer.	Compt.	Node Classification	Recommendation
GNN Compression	GCN, GAT, GIN	Train. & Infer.	Compt.	Node Classification	Dynamic Graphs

Table 3: Comparison among algorithmic acceleration methods from multiple aspects.

ducted on spatial-based GNNs, e.g., GCNs, to capture neighboring connection and representation in a spatial dimension. A typical *sparsification* method can be regarded as a *sampling* method conditionally by viewing each edge as the sampling target. Moreover, *sparsification* methods have wider backbones than *sampling* methods for application. *Partition* methods are mainly applied to special GNNs that are spatial-partible for subgraphs generation. Existing *simplification* methods merely design simplified GCNs for application, owing to straightforward propagation rules and the widespread usage of GCNs. *Compression* methods have been previously used for DNNs acceleration [Cheng *et al.*, 2017], in which techniques that used, such as quantification, can also be deployed on GNNs and most variants with modifications.

• ***Acceleration phase*** denotes which phases in GNN execution are accelerated. Since training is the most time-consuming phase, accelerating training is the primary objective for all these methods. Herein, we highlight some special cases. *Sparsification* methods using learnable modules like UGS [Chen *et al.*, 2021] and AdaptiveGCN [Li *et al.*, 2021] adopt sparsified graphs in inference to yield speedup. *Simplification* methods generally benefit both training and inference in speed, since the cost of training and inference in a simplified model is always saved. *Compression* methods provide model-level optimizations in terms of model weights and structures. Same as *simplification* methods, the benefit of GNN *compression* is favorable to both training and inference.

• ***Optimization objective*** denotes which objectives are optimized to yield a speedup. Most methods reduce the computation cost (abbreviated as **Compt.**) to accelerate GNN execution, such as *sampling* methods. Generally, the cost of aggregating neighbor representation in a full-batch manner largely depends on the number of edges. *Sparsification* methods drop useless edges in a graph to reduce memory access (abbreviated as **Mem.**) cost, thus yielding a speedup to a certain degree. *Partition* methods reduce communication cost (abbreviated as **Com.**) by minimizing cross-partition edges. Moreover, memory-aware *partition* methods achieve load balance according to a reasonable allocation of memory, which benefits GNN execution in terms of speed.

• ***General and special application*** denotes the general application and the special application respectively. Generally, all GNN based methods can resolve graph-related tasks such as (semi-supervised) node classification. Specially, *sampling* methods, e.g., AS-GCN[Huang *et al.*, 2018], GraphSAINT[Zeng *et al.*, 2020], can be leveraged to eliminate variance that introduced by probabilistic sampling. By regarding redundant edges in a graph as noise, *sparsification* methods such as NeuralSparse[Zheng *et al.*, 2020a] can learn a specific strategy to remove task-irrelevant edges, achieving an effect of denoising. Instead of randomly generating subgraphs, *partition* methods can divide a graph into many smaller ones by using clustering algorithms, where size of each cluster is similar for load balance. *Simplification* methods such as LightGCN [He *et al.*, 2020] fuse a simplified GNN models with processes such as collaborative filtering, which is designed for a recommendation task. By adding virtual edges to graphs in a teacher model and a student model, *compression* methods such as KD-GCN [Yang *et al.*, 2020] can extend the process of KD to dynamic graph learning.

5 Summary and Future Prospects

This paper provides a comprehensive survey on algorithmic acceleration methods for GNNs, in which methods in existing literature are systematically classified, discussed, and compared according to the proposed taxonomy. We believe the execution of GNNs can be promoted to gain higher efficiency via graph and model level optimizations, benefiting graph-related tasks in diverse platforms. Despite recent success and

great leap of GNN acceleration methods, there are still challenges to be solved in this research field. We thereby suggest some promising prospects for future research as follows.

• ***Acceleration for dynamic graphs:*** most acceleration methods adopt static graphs for research. A dynamic graph, however, is more flexible in topology and feature spaces than a static one, making it hard to apply these methods to dynamic graphs directly. Compression methods like KD-GCN [Yang *et al.*, 2020] and Binarized DGCNN [Bahri *et al.*, 2021] utilize a special module to extend the use to dynamic graphs, providing a nascent exemplar of dynamic graphs acceleration.

• ***Hardware-friendly algorithms:*** hardware-friendly algorithms benefit the model (or algorithm) execution on general platforms by leveraging hardware features. Recent literature [Liu *et al.*, 2021c] that targets to bridge the gap between graph sampling algorithms and the hardware feature, utilizes locality-aware optimizations to yield a considerable speedup in graph sampling. However, this raises the question of what characteristics should be carefully considered to design a hardware-friendly algorithm for GNN acceleration.

• ***Algorithm and hardware co-design***: different to domain-specific hardware accelerators for GNNs, e.g., HyGCN [Yan *et al.*, 2020c] directly tailoring hardware datapath to GNNs based on the execution-semantic characterization for GNNs [Yan *et al.*, 2020b], algorithm and hardware co-design explores the design space with both algorithm and hardware awareness. Taking a productive co-design in a related field (i.e., graph processing) as an example, GraphDynS [Yan *et al.*, 2019] first optimizes the execution semantic of graph traversal algorithms and then tailors its hardware datapath to the optimized execution semantic. Similarly, in GNN acceleration, a synergy effect on optimization can be achieved by a simultaneous design of algorithm and hardware efforts in general. However, to our knowledge, there has been rare existing work on this perspective so far.

Acknowledgments

This work was supported by the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDC05000000), National Natural Science Foundation of China (Grant No. 61732018 and 61872335), Austrian-Chinese Cooperative R&D Project (FFG and CAS) (Grant No. 171111KYSB20200002), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-029), and CAS Project for Youth Innovation Promotion Association.

References

[Abadal *et al.*, 2021] Sergi Abadal, Akshay Jain, Robert Guirado, and et al. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)*, 54(9):1–38, 2021.

[Arka *et al.*, 2021] Aqeeb Iqbal Arka, Biresh Kumar Joardar, Janardhan Rao Doppa, and et al. Dare: Droplayer-aware manycore reram architecture for training graph neural networks. In *ICCAD*, pages 1–9, 2021.

[Bahri *et al.*, 2021] Mehdi Bahri, Gaétan Bahl, and Stefanos Zafeiriou. Binary graph neural networks. In *CVPR*, pages 9492–9501, 2021.

[Battaglia *et al.*, 2018] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, and et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[Bojchevski *et al.*, 2020] Aleksandar Bojchevski, Johannes Klicpera, Bryan Perozzi, and et al. Scaling graph neural networks with approximate pagerank. In *SIGKDD*, pages 2464–2473, 2020.

[Boyd *et al.*, 2011] Stephen Boyd, Neal Parikh, and Eric Chu. *Distributed optimization and statistical learning via the alternating direction method of multipliers*. Now Publishers Inc, 2011.

[Buluç *et al.*, 2016] Aydın Buluç, Henning Meyerhenke, Ilya Safro, and et al. Recent advances in graph partitioning. *Algorithm engineering*, pages 117–158, 2016.

[Chen *et al.*, 2018a] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. In *ICML*, pages 941–949, 2018.

[Chen *et al.*, 2018b] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: Fast learning with graph convolutional networks via importance sampling. In *ICLR*, 2018.

[Chen *et al.*, 2021] Tianlong Chen, Yongduo Sui, Xuxi Chen, and et al. A unified lottery ticket hypothesis for graph neural networks. In *ICML*, pages 1695–1706, 2021.

[Cheng *et al.*, 2017] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[Chiang *et al.*, 2019] Wei-Lin Chiang, Xuanqing Liu, Si Si, and et al. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *SIGKDD*, pages 257–266, 2019.

[Eppstein *et al.*, 1997] David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *JACM*, 44(5):669–696, 1997.

[Fey and Lenssen, 2019] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[Hamilton *et al.*, 2017] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NeurIPS*, pages 1025–1035, 2017.

[Han *et al.*, 2021] Li Han, Yan Mingyu, Lü Zhengyang, Li Wenming, Ye Xiaochun, Fan Dongrui, and Tang Zhimin. Survey on graph neural network acceleration architectures. *Journal of Computer Research and Development*, 58(6):1204, 2021.

[He *et al.*, 2020] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *SIGIR*, pages 639–648, 2020.

[Hinton *et al.*, 2015] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[Hu *et al.*, 2020a] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[Hu *et al.*, 2020b] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous graph transformer. In *WWW*, pages 2704–2710, 2020.

[Huang *et al.*, 2018] Wenbing Huang, Tong Zhang, Yu Rong, and et al. Adaptive sampling towards fast graph representation learning. *Advances in Neural Information Processing Systems*, 31:4558–4567, 2018.

[Karypis and Kumar, 1998] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[Kipf and Welling, 2017] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[Lamb *et al.*, 2020] Luís C. Lamb, Artur S. d'Avila Garcez, Marco Gori, Marcelo O. R. Prates, Pedro H. C. Avelar, and Moshe Y. Vardi. Graph neural networks meet neural-symbolic computing: A survey and perspective. In *IJCAI*, pages 4877–4884, 2020.

[Li *et al.*, 2020] Jiayu Li, Tianyun Zhang, Hao Tian, Shengmin Jin, Makan Fardad, and Reza Zafarani. Sgcn: A graph sparsifier based on graph convolutional networks. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 275–287. Springer, 2020.

[Li *et al.*, 2021] Dongyue Li, Tao Yang, Lun Du, and et al. Adaptivegcn: Efficient gcn through adaptively sparsifying graphs. In *CIKM*, pages 3206–3210, 2021.

[Lin *et al.*, 2020] Zhiqi Lin, Cheng Li, Youshan Miao, and et al. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *SoCC*, pages 401–415, 2020.

[Liu *et al.*, 2021a] Tianfeng Liu, Yangrui Chen, Dan Li, and et al. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *arXiv preprint arXiv:2112.08541*, 2021.

[Liu *et al.*, 2021b] Xin Liu, Mingyu Yan, Lei Deng, and et al. Sampling methods for efficient training of graph convolutional networks: A survey. *IEEE/CAA Journal of Automatica Sinica*, 9(2):205–234, 2021.

[Liu *et al.*, 2021c] Xin Liu, Mingyu Yan, Shuhan Song, and et al. Gnnsampler: Bridging the gap between sampling algorithms of gnn and hardware. *arXiv preprint arXiv:2108.11571*, 2021.

[Mao *et al.*, 2021] Kelong Mao, Jieming Zhu, Xi Xiao, Biao Lu, Zhaowei Wang, and Xiuqiang He. Ultragcn: Ultra simplification of graph convolutional networks for recommendation. In *CIKM*, pages 1253–1262, 2021.

[Md *et al.*, 2021] Vasimuddin Md, Sanchit Misra, Guixiang Ma, and et al. Distgnn: Scalable distributed training for large-scale graph neural networks. *arXiv preprint arXiv:2104.06700*, 2021.

[Pope *et al.*, 2019] Phillip E Pope, Soheil Kolouri, Mohammad Rostami, Charles E Martin, and Heiko Hoffmann. Explainability methods for graph convolutional neural networks. In *CVPR*, pages 10772–10781, 2019.

[Rong *et al.*, 2020] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. In *ICLR*, 2020.

[Scarselli *et al.*, 2008] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[Srinivasa *et al.*, 2020] Rakshith S Srinivasa, Cao Xiao, Lucas Glass, and et al. Fast graph attention networks using effective resistance based graph sparsification. *arXiv preprint arXiv:2006.08796*, 2020.

[Tailor *et al.*, 2021] Shyam Anil Tailor, Javier Fernández-Marqués, and Nicholas Donald Lane. Degree-quant: Quantization-aware training for graph neural networks. In *ICLR*, 2021.

[Veličković *et al.*, 2018] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *ICLR*, 2018.

[Wang *et al.*, 2019a] Minjie Wang, Da Zheng, Zihao Ye, and et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[Wang *et al.*, 2019b] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. Neural graph collaborative filtering. In *SIGIR*, pages 165–174, 2019.

[Wang *et al.*, 2021a] Junfu Wang, Yunhong Wang, Zhen Yang, and et al. Bi-gcn: Binary graph convolutional network. In *CVPR*, pages 1561–1570, 2021.

[Wang *et al.*, 2021b] Shoujin Wang, Liang Hu, Yan Wang, and et al. Graph learning based recommender systems: A review. In *IJCAI*, pages 4644–4652, 2021.

[Wu *et al.*, 2019] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *ICML*, pages 6861–6871, 2019.

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

[Xu *et al.*, 2019] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *ICLR*, 2019.

[Yan *et al.*, 2019] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, et al. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 615–628, 2019.

[Yan *et al.*, 2020a] Bencheng Yan, Chaokun Wang, Gaoyang Guo, and Yunkai Lou. Tinygnn: Learning efficient graph neural networks. In *SIGKDD*, pages 1848–1856, 2020.

[Yan *et al.*, 2020b] Mingyu Yan, Zhaodong Chen, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Characterizing and understanding gcns on gpu. *IEEE Computer Architecture Letters*, 19(1):22–25, 2020.

[Yan *et al.*, 2020c] Mingyu Yan, Lei Deng, Xing Hu, and et al. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

[Yang *et al.*, 2020] Yiding Yang, Jiayan Qiu, Mingli Song, Dacheng Tao, and Xinchao Wang. Distilling knowledge from graph convolutional networks. In *CVPR*, pages 7074–7083, 2020.

[Yang *et al.*, 2021] Cheng Yang, Jiawei Liu, and Chuan Shi. Extract the knowledge of graph neural networks and go beyond it: An effective knowledge distillation framework. In *WWW*, pages 1227–1237, 2021.

[Ying *et al.*, 2019] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. *Advances in neural information processing systems*, 32:9240, 2019.

[Zeng *et al.*, 2019] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Accurate, efficient and scalable graph embedding. In *IPDPS*, pages 462–471. IEEE, 2019.

[Zeng *et al.*, 2020] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graph-SAINT: Graph sampling based inductive learning method. In *ICLR*, 2020.

[Zhang *et al.*, 2019] Chuxu Zhang, Dongjin Song, Chao Huang, and et al. Heterogeneous graph neural network. In *SIGKDD*, pages 793–803, 2019.

[Zhang *et al.*, 2020] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE TKDE*, 2020.

[Zhang *et al.*, 2021] Ziwei Zhang, Xin Wang, and Wenwu Zhu. Automated machine learning on graphs: A survey. In *IJCAI*, pages 4704–4712, 2021.

[Zhao *et al.*, 2021] Tong Zhao, Yozen Liu, Leonardo Neves, and et al. Data augmentation for graph neural networks. In *AAAI*, pages 11015–11023, 2021.

[Zheng *et al.*, 2020a] Cheng Zheng, Bo Zong, Wei Cheng, and et al. Robust graph representation learning via neural sparsification. In *ICML*, pages 11458–11468, 2020.

[Zheng *et al.*, 2020b] Da Zheng, Chao Ma, Minjie Wang, and et al. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44, 2020.