

# A Survey of Graph Pre-processing Methods: From Algorithmic to Hardware Perspectives

ZHENGYANG LV, MINGYU YAN\*, and XIN LIU, SKLP, ICT, CAS, China and UCAS, China

MENGYAO DONG, ShanghaiTech Univ., China and SHIC, China

XIAOCHUN YE, DONGRUI FAN, and NINGHUI SUN, ICT, CAS, China

Graph-related applications have experienced significant growth in academia and industry, driven by the powerful representation capabilities of graph. However, efficiently executing these applications faces various challenges, such as load imbalance, random memory access, etc. To address these challenges, researchers have proposed various acceleration systems, including software frameworks and hardware accelerators, all of which incorporate graph pre-processing (GPP). GPP serves as a preparatory step before the formal execution of applications, involving techniques such as sampling, reorder, etc. However, GPP execution often remains overlooked, as the primary focus is directed towards enhancing graph applications themselves. This oversight is concerning, especially considering the explosive growth of real-world graph data, where GPP becomes essential and even dominates system running overhead. Furthermore, GPP methods exhibit significant variations across devices and applications due to high customization. Unfortunately, no comprehensive work systematically summarizes GPP. To address this gap and foster a better understanding of GPP, we present a comprehensive survey dedicated to this area. We propose a double-level taxonomy of GPP, considering both algorithmic and hardware perspectives. Through listing relevant works, we illustrate our taxonomy and conduct a thorough analysis and summary of diverse GPP techniques. Lastly, we discuss challenges in GPP and potential future directions.

**CCS Concepts:** • General and reference → Surveys and overviews; • Theory of computation → Graph algorithms analysis; • Hardware → Hardware accelerators; • Computer systems organization → Architectures.

**Additional Key Words and Phrases:** Graph pre-processing, traditional graph computing, graph neural networks, hardware acceleration

## ACM Reference Format:

Zhengyang Lv, Mingyu Yan, Xin Liu, Mengyao Dong, Xiaochun Ye, Dongrui Fan, and Ninghui Sun. 2023. A Survey of Graph Pre-processing Methods: From Algorithmic to Hardware Perspectives. 1, 1 (September 2023), 38 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

Graph processing applications have garnered significant attention for their ability to provide valuable insights from graph data. In various real-world scenarios, data can be effectively represented using graph structures, with social networks being a prime example [103]. For instance, Figure 1

---

Authors' addresses: Zhengyang Lv, lvzhengyang19@mails.ucas.ac.cn; Mingyu Yan, yanmingyu@ict.ac.cn; Xin Liu, liuxin196@mails.ucas.ac.cn, SKLP, ICT, CAS, No.6 Kexueyuan South Road Zhongguancun, Haidian District, Beijing, China, 100190 and UCAS, Beijing, China; Mengyao Dong, dongmy@shanghaitech.edu.cn, ShanghaiTech Univ., Shanghai, China and SHIC, Shanghai, China; Xiaochun Ye, yexiaochun@ict.ac.cn; Dongrui Fan, Fandr@ict.ac.cn; Ninghui Sun, snh@ict.ac.cn, ICT, CAS, Beijing, China.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2023 Association for Computing Machinery.

XXXX-XXXX/2023/9-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

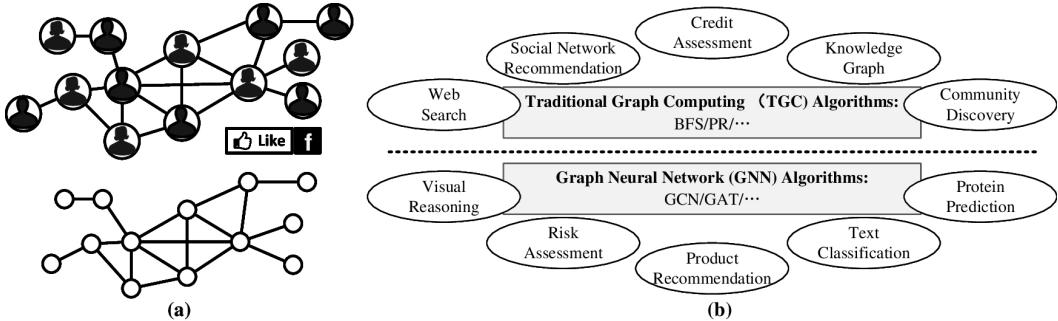


Fig. 1. Graph processing algorithms as enablers of many applications that depend on graph data: (a) Graphs abstracted from Facebook networks, where vertices denote users and edges denote users’ relationships; (b) Applications of graph processing algorithms.

(a) depicts the abstraction of the social network of Facebook as a graph. There are two most widely used types of graph processing applications: traditional graph computing (TGC), which includes algorithms like breadth-first search (BFS), pagerank (PR), and others; and graph neural network (GNN) such as graph convolution network (GCN) and graph attention network (GAT). These graph processing algorithms find extensive use in various scenarios, including social network recommendation [130], knowledge graph analysis [79], protein prediction [37], visual reasoning [117], and more. To handle the exponentially growing scale of graph data, these algorithms have become increasingly popular and are widely deployed in diverse data centers, such as Google-Maps [28], Microsoft-Academic-Graph [46], Alibaba’s E-commerce platform [112], Baidu-Maps [31], etc.

The execution of graph processing algorithms faces several challenges, and numerous efforts have been designed to alleviate these issues. *Firstly*, in TGC algorithms, the execution behavior, including factors like resource utilization and operation sequencing, often exhibits irregularities. These irregularities arise from the irregular topology of graphs, leading to irregular workloads, memory access, and communication [13]. To tackle these challenges, various frameworks based on general hardware platforms (CPU & GPU) have been proposed, such as GraphChi [55] and CuSha [54]. Also, custom architectures have been developed for further acceleration, such as Graphicionado [42] based on ASIC (Application-Specific Integrated Circuit), ForeGraph [25] based on FPGA (Field Programmable Gate Array), and GraphR [89] based on PIM (Processing-In-Memory). *Secondly*, GNN algorithms exhibit a combination of irregular and regular execution behaviors [107, 110], as the adding of neural networks (NNs) to transform high-dimensional feature vectors for each vertex. To address both regular and irregular characteristics in GNNs, several dedicated acceleration platforms have been proposed, such as HyGCN [108] based on ASIC and GraphACT [118] based on FPGA. *Thirdly*, the explosive growth of graph data has prompted exploration into parallel processing for large-scale graphs. TGC parallel frameworks include Pregel [74], GraphLab [70], and GNN parallel frameworks include DistDGL [132], PaGraph [63], etc.

Graph processing execution heavily relies on a critical operation—graph pre-processing (GPP). For example, GraphChi [55], Graphicionado [42], GraphDynS [109], FPGP [24], and HyGCN [108] utilize reorganization techniques to pre-split the graph data into shards, enabling contiguous memory access and enhancing performance. In parallel graph processing systems like Pregel [74], GraphLab [70], DistDGL [132], and PaGraph [63], graph partition is performed in advance to divide large-scale graph data into multiple subgraphs and assign them to multiple processors/machines, achieving load balance and minimizing communication overhead. To facilitate the efficient training of GNN in parallel, PaGraph [63] and DistDGL [132] create mini-batches using sampling techniques. GraphACT [118] and GCNIinfer [120] pre-merge common neighbors to reduce subsequent

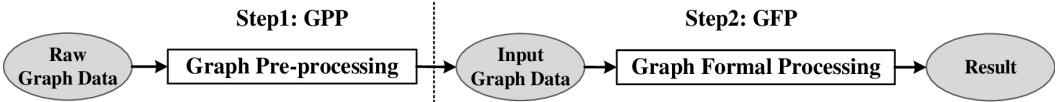


Fig. 2. Two main steps in graph processing systems: graph pre-processing (GPP) and graph formal processing (GFP).

redundant operations. Therefore, GPP is critical for efficiently executing graph processing algorithms, benefiting a wide range of graph processing systems, including single-machine graph processing frameworks, distributed graph processing frameworks, graph processing accelerators, etc.

To provide clarity, we abstract a typical graph processing system into two main steps: graph pre-processing (GPP) and graph formal processing (GFP), as illustrated in Figure 2. During the GPP step, various operations are performed on the raw graph data to prepare the input dataset for subsequent execution of graph processing algorithms. In the GFP step, the computing unit loads the pre-processed data and executes the graph processing algorithm to obtain the final result. It is worth noting that the choice of GPP methods depends on the characteristics of the raw graph dataset, as well as the execution platforms. For example, partition is employed to manage large-scale graph data in parallel systems, with researches like DistDGL [132] using CPU-clusters and PaGraph [63] using multi-GPUs. GraphACT [118] employs reconstruction methods to reduce redundant computing on FPGA, achieving high performance and energy efficiency. Overall, GPP offers two main benefits: a) reducing overhead in computing, storage, and communication; b) meeting the execution requirements of various algorithms on devices with limited resources.

Unfortunately, the GPP overhead has become increasingly significant due to the explosive growth of graph data. Next, we give the following examples to visualize the importance of GPP via numerical comparisons. In Graph500 competition<sup>1</sup>, Fugaku [81], a petascale supercomputer, exhibits the high GPP time (C\_TIME) of 390 seconds, a striking 1560× contrast to the BFS execution time of 0.25 seconds. In Gorder [102], graph reorder on a large Twitter dataset takes 1.5 hours, while PageRank completes 100 iterations in just 13.65 minutes. Therefore, if the input graph is not frequently reused, the significant GPP time for large datasets may not be a worthwhile investment. Similarly, in Graphite [38], when executing GraphSAGE, the sampling time comprises over 80% of the total training time. These examples highlight the significance of reducing GPP overhead to improve the overall execution efficiency of graph processing systems.

Based on the preceding analysis, two significant conclusions emerge, highlighting the pressing need of GPP surveys. Firstly, GPP is crucial for efficient graph processing. Secondly, the GPP overhead is becoming significant and non-negligible, necessitating the reduction of GPP overhead. As a result, the field of GPP holds immense potential for further exploration, and more in-depth studies are required to fully explore the optimization possibilities offered by GPP.

Nevertheless, there is a lack of comprehensive review of GPP techniques across the field, despite some studies analyzing individual GPP methods [2, 3, 21, 66]. This gap hinders a holistic understanding of the potential optimizations that can be achieved through GPP. In Table 1, we present a list of related surveys in the field of graph processing and involving GPP methods in these works. Some surveys have explored both TGC acceleration and GNN acceleration techniques, with some touching on GPP methods. For example, surveys on GPU and FPGA-based TGC [11, 88] involve partition techniques for handling large graphs. Other works [40, 45, 76, 83] analyze static and dynamic graph partition in distributed systems and memory-based graph processing systems. Recent surveys [1, 16, 59, 62, 67, 86, 87, 97, 127] extensively cover GNN acceleration and describe the significance of GPP steps in GNN execution. However, these surveys still concentrate on analyzing the optimization of GFP step, with GPP not being their primary focus.

<sup>1</sup>[https://graph500.org/?page\\_id=834](https://graph500.org/?page_id=834)

Table 1. Surveys about graph processing acceleration and the GPP methods involved

Survey (Year)	Domain	GPP
• Graph Processing on GPUs: A Survey (2018) [88]	TGC GPU	Partition Reorganization
• Graph Processing on FPGAs: Taxonomy, Survey, Challenges (2019) [11]	TGC FPGA	Partition Reorganization
• A Survey on Graph Processing Accelerators: Challenges and Opportunities (2019) [40]	TGC Hardware	Partition Reorder
• Graph Processing and Machine Learning Architectures with Emerging Memory Technologies: A Survey (2021) [83]	TGC & GNN PIM	Reorganization
• Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing (2015) [76]	TGC	Partition
• Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges (2018) [45]	Parallel System	Reorganization
• A Survey of Field Programmable Gate Array (FPGA)-based Graph Convolutional Neural Network Accelerators: Challenges and Opportunities (2022) [59]	GNN FPGA	Partition Sampling Quantification
• Bottleneck Analysis of Dynamic Graph Neural Network Inference on CPU and GPU (2022) [16]	Dynamic GNN CPU & GPU	Partition Sampling
• Scalable Graph Neural Network Training: The Case for Sampling (2021) [86]	GNN Algorithm	Partition Sampling
• A Comprehensive Survey on Distributed Training of Graph Neural Networks (2022) [62]	GNN CPU & GPU	Partition Sampling
• Distributed Graph Neural Network Training: A Survey (2022) [87]		
• The Evolution of Distributed Systems for Graph Neural Networks and their Origin in Graph Processing and Deep Learning: A Survey (2023) [97]	GNN Parallel System	Partition Sampling
• Survey on Graph Neural Network Acceleration: An Algorithmic Perspective (2022) [67]	GNN Algorithm	Many
• Computing Graph Neural Networks: A Survey from Algorithms to Accelerators (2021) [1]	GNN Algorithm	Many
• A Survey on Graph Neural Network Acceleration: Algorithms, Systems, and Customized Hardware (2023) [127]	Hardware	
• Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations (2018) [21]	Graph Algorithm	Quantification
• Streaming Graph Partitioning: An Experimental Study (2018) [2]	GNN Algorithm	
• A Survey of Current Challenges in Partitioning and Processing of Graph Structured Data in Parallel and Distributed Systems (2020) [3]		Partition
• Sampling Methods For Efficient Training of Graph Convolutional Networks: A Survey (2021) [66]	GNN Algorithm	Sampling

To harness the full potential of GPP in graph processing, it is crucial to perform both hardware and algorithm optimizations. However, there is a gap between hardware acceleration and algorithm optimization in GPP. Existing research predominantly focuses on hardware acceleration of GFP, with limited attention given to GPP, or it may only analyze individual GPP techniques at the algorithmic level. This article primarily aims to bridge this gap by providing a systematic and comprehensive summary and analysis of GPP methods, encompassing both algorithmic and hardware perspectives. We are honored to present a comprehensive overview of GPP methods, with the aim of contributing to the advancement of GPP and offering a reference for further research in this area. Our work may provide valuable insights for the future optimization of GPP execution and graph processing acceleration. Our contributions are as follows:

**Review:** We review the challenges associated with graph processing execution, considering the aspects of computing, storage, and communication. We highlight the significance of GPP for optimizing execution through relevant examples.

**Taxonomy:** We classify existing GPP methods and propose a double-level taxonomy from both algorithmic and hardware perspective. The algorithmic categories include graph representation optimization and data representation optimization. The hardware categories encompass efficient computing, storage, and communication.

**Analysis:** We provide detailed introductions to the existing GPP methods in accordance with the proposed taxonomy. Specifically, we list and analyze related works from both algorithmic and hardware perspectives.

**Comparison:** We offer a comprehensive summary and comparison of existing GPP methods considering both algorithmic and hardware aspects, allowing for a better understanding of their strengths and weaknesses.

**Discussion:** We discuss the challenges associated with GPP such as high overhead, accuracy loss, etc. Finally, we outline potential research directions for future exploration.

The rest of this paper is organized as follows: Section 2 provides preliminary GPP information, covering graph concepts and algorithms. Section 3 explores execution challenges of graph processing and demonstrates how GPP can address them. Section 4 presents our double-level GPP taxonomy based on algorithmic optimization factors and hardware optimization effects. Sections 5 and 6 respectively analyze GPP methods with examples from algorithmic and hardware perspectives. Section 7 offers a comprehensive summary and comparison. Section 8 discusses prevailing GPP bottlenecks and potential research directions. Finally, Section 9 concludes our work.

## 2 PRELIMINARY

In this section, we will begin by introducing the fundamental concepts of graphs that are used throughout the subsequent sections, covering graph representation and storage format. Then, we will outline the two types of graph processing algorithms: TGC algorithms and GNN algorithms. To facilitate a comprehensive analysis of the execution process for these algorithms, we will also present the programming models that are commonly employed.

Table 2. Notations and corresponding descriptions used in this work.

Notations	Descriptions
$G = (V, E)$	A graph and its vertex sets $V$ and edge sets $E$ .
$n, m$	The number of vertices and edges, $n =  V , m =  E $ .
$v, e_{ij}$	A vertex $v \in V$ , an edge from $i$ to $j$ , $e_{ij} \in E$ .
$N(v), SN(v)$	Original and sampled neighbourhood set of vertex $v$ .
$\mathbf{A}, \tilde{\mathbf{A}}$	The original and normalized adjacency matrix of graph.
$\mathbf{D}, \tilde{\mathbf{D}}$	The degree matrix of $\mathbf{A}$ and $\tilde{\mathbf{A}}$ .
$\mathbf{W}_V, \mathbf{W}_E$	The vertex and edge weight matrix of graph.
$d$	The dimension of a vertex feature vector or hidden vertex feature vector.
$\mathbf{X}, \mathbf{H}$	The feature matrix and the hidden feature matrix of graph.
$\mathbf{x}_v, \mathbf{h}_v$	The feature vector and the hidden feature vector of the vertex $v$ , $\mathbf{x}_v \in \mathbf{R}^d, \mathbf{h}_v \in \mathbf{R}^d$ .
$L, l$	The number of GNN layers, and the index of each layer.

### 2.1 Graph Representation and Storage Format

As illustrated in Table 2, a graph is represented as  $G = (V, E)$ , where the set of vertices  $V = \{v_0, v_1, \dots, v_{n-1}\}$ , and the set of edges  $E = \{e_0, e_1, \dots, e_{m-1}\}$ .  $v_i \in V$  denotes a vertex in the graph, and  $e_{ij} = (v_i, v_j) \in E$  denotes an edge pointing from  $v_i$  (source) to  $v_j$  (destination). The vertex relationship is represented by an adjacency matrix  $\mathbf{A}$  with size  $n \times n$ . If  $e_{ij} \in E$ , then  $A_{ij} = 1$ , otherwise  $A_{ij} = 0$ .

$= 0$ . The degree matrix  $\mathbf{D}$  is a diagonal matrix, where  $D_{ii}$  represents the degree of vertex  $i$ . The neighbourhood set of vertex  $v$  is defined as  $N(v) = \{u \in V | (v, u) \in E\}$ . Graph vertex attributes are represented as a multidimensional feature matrix  $\mathbf{X} \in \mathbf{R}^{n \times d}$ .  $\mathbf{x}_v = (x_0, x_1, \dots, x_{d-1})$  is the feature vector of the vertex  $v$ . For GNN,  $\mathbf{h}_v^{(l)}$  represents the hidden feature vector of layer  $l$  of vertex  $v$ ,  $\mathbf{W}_v^{(l)}$  and  $\mathbf{W}_E^{(l)}$  represent the feature matrix of layer  $l$ .

There are three main storage formats for representing the non-zero elements in the matrix: coordinate list (COO), compressed sparse row (CSR), and compressed sparse column (CSC). As depicted in Figure 3, the COO format stores the row index, column index, and corresponding value (e.g., features, weights) of each non-zero element in separate arrays. Although simple and intuitive, the COO format contains redundant elements. To address this, the CSR and CSC formats were introduced to compress the redundant data. In the CSR format, the non-zero elements are stored in three arrays using row compression. The offset array stores the position of the first non-zero element in each row within the column array and the total number of non-zero elements. Similarly, the CSC format employs column compression.

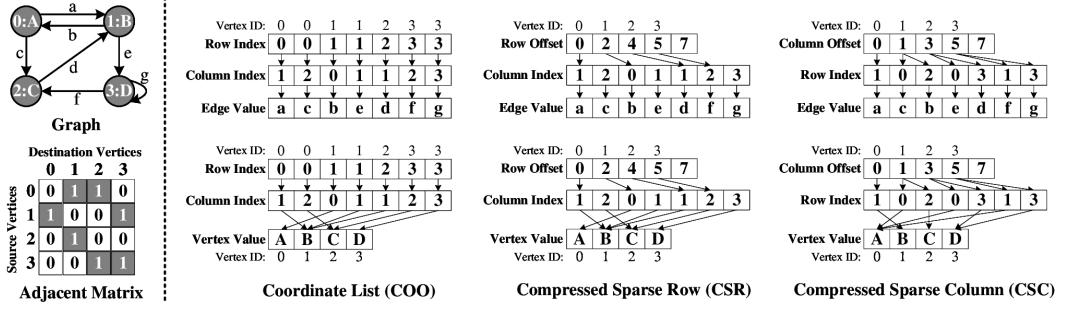


Fig. 3. Storage format of adjacency matrix: The value includes the weights and features associated with vertices or edges. In TGC, one-dimensional features are typically employed, while GNNs commonly utilize high-dimensional feature vectors.

## 2.2 Graph Processing Algorithms

In this subsection, we will introduce two types of commonly used graph processing algorithms: TGC algorithms and GNN algorithms. We will explore typical examples and programming models related to each algorithm type. These programming models offer abstractions and guidelines for developing efficient graph processing algorithms. By providing this comprehensive introduction, our aim is to offer a clearer understanding of the execution process of these graph processing algorithms, enabling better acceleration and optimization strategies.

**2.2.1 Traditional Graph Computing (TGC) Algorithms.** The execution of such algorithms involves iterative updates of vertex information. Based on whether each iteration traverses all vertices, TGC algorithms can be classified into two categories: stationary graph algorithms and non-stationary graph algorithms [45, 53]. The stationary graph algorithms update all vertices in each iteration and rely on the current state of all vertices to compute the next state. Typical algorithms include PR, diameter estimation (DE), random walk with restart (RWR), etc. The non-stationary graph algorithms optimize the update process by considering vertex dependencies and update only the necessary portions of the graph. Typical algorithms include BFS, depth-first search (DFS), single-source shortest path (SSSP), etc.

The programming models for TGC play a crucial role in efficiently processing large-scale graphs. As depicted in Figure 4, three common TGC programming models are the vertex-centric programming model (VCPM) [74], the edge-centric programming model (ECPM) [85], and the gather-apply-scatter programming model (GAS) [39]. VCPM enables parallel execution of vertex processing and

Vertex-Centric Programming Model (VCPM)	Edge-Centric Programming Model (ECPM)	Gatter-Apply-Scatter Programming Model (GAS)
<pre> <b>while</b> not done <b>do</b>   <b>for</b> each vertex <math>v</math> that need to scatter updates     <i>Vertex_Scatter</i>(<math>v</math>): send <math>v</math> to neighbours   <b>for</b> each vertex <math>u</math> that have updates     <i>Vertex_Gather</i>(<math>u</math>): update vertex <math>u</math>.dest </pre>	<pre> <b>while</b> not done <b>do</b>   <b>for</b> each edge <math>e</math>.src that has scatter updates     <i>Edge_Scatter</i>(<math>e</math>): send <math>u</math> to vertex <math>e</math>.dest   <b>for</b> each vertex <math>u</math> that have updates     <i>Edge_Gather</i>(<math>u</math>): update vertex <math>u</math>.dest </pre>	<b>for</b> each active vertex $v$ <b>Define:</b> vertex and edge data $D_v, D_{(v,u)}$ <i>Gather</i> ( $D_v, D_{(v,u)}, D_u$ ) $\rightarrow$ $Acc$ //gather neighbours of $v$ <i>Process</i> ( $Acc1, Acc2$ ) $\rightarrow$ $Acc$ <i>Apply</i> ( $D_v, Acc$ ) $\rightarrow$ $D_v^{new}$ <i>Scatter</i> ( $D_v^{new}, D_{(v,u)}, D_u$ ) $\rightarrow$ ( $D_{(v,u)}, Acc$ ) //scatter neighbours of $v$

Fig. 4. Programming mode of VCPM, ECPM [85] and GAS [39]: VCPM starts from each source vertex or active vertex, transmitting information to neighbors. ECPM starts from each edge, collecting information from source to destination vertices. GAS iterates through three stages, where Gather collects neighbor information from incoming edges, Apply updates central vertex attributes based on the collected information, and Scatter updates the neighbor vertex information of outgoing edges with the new value.

is commonly used in parallel systems. ECPM ensures continuous access to edge data, enhancing spatial locality. GAS offers flexibility, finding widespread usage in various software frameworks and accelerator designs. Developers can leverage the strengths of each approach to efficiently process graph data based on algorithm characteristics, underlying computing infrastructure and scenario requirements.

**2.2.2 Graph Neural Network (GNN) Algorithms.** GNNs are a powerful class of models that integrate graph structures and NNs to effectively model and learn attributes and connections between vertices. GNN algorithms process input information iteratively layer by layer, to calculate information relevant to specific vertices. Several GNN models have been developed for diverse scenarios and can be categorized into four groups: convolutional graph neural network (ConvGNN), recurrent graph neural network (RecGNN), graph auto-encoder (GAE), and spatial-temporal graph neural network (STGNN) [103]. Among these, two ConvGNN models stand out as particularly prevalent and widely used: graph convolutional neural network (GCN) [80] and graph attention network (GAT) [98]. Currently, the majority of mainstream frameworks and accelerators are specifically optimized to enhance the performance of GCN and GAT.

A GCN can be generally abstracted in Equations 1 & 2. During the aggregate stage, the GCN traverses the entire graph to collect and aggregate neighbor information for each vertex, resulting in an aggregated intermediate feature for the current layer  $\mathbf{h}_{N(v)}^{(l)}$ , which is called hidden feature. In the update stage, the intermediate feature, along with the output feature from the previous layer  $\mathbf{h}_{N(v)}^{(l)}$ , is combined to update the output feature vector of the current layer  $\mathbf{h}_v^{(l)}$ .

$$\mathbf{h}_{N(v)}^{(l)} = \text{Aggregate}^{(l)}(\{\mathbf{h}_u^{(l-1)} : u \in N(v)\}) \quad (1)$$

$$\mathbf{h}_v^{(l)} = \text{Update}^{(l)}(\{\mathbf{h}_v^{(l-1)}, \mathbf{h}_{N(v)}^{(l)}\}) \quad (2)$$

Similar to a NN, a GNN consists of two parts: training and inference. During inference, forward propagation takes place, where aggregate and update processes alternate until reaching the maximum number of execution layers  $L$ . In the training process, forward and backward propagation are iteratively performed to adjust the training parameters towards minimizing the loss function  $\mathcal{L}$ . The loss function  $\mathcal{L}$  is utilized to measure the discrepancy between the predicted value and the true value. Generally, aggregate and update can be achieved by matrix multiplication, including SPMM or sparse-dense matrix multiplication (SPDMM). Here we give the hidden layer calculation formulas of GCN and GAT.

$$\mathbf{H}^l = \sigma \left( \widetilde{\mathbf{D}}^{-\frac{1}{2}} \widetilde{\mathbf{A}} \widetilde{\mathbf{D}}^{-\frac{1}{2}} \mathbf{H}^{l-1} \mathbf{W}^{l-1} \right) \quad (3)$$

$$\mathbf{h}_v^{(l)} = \sigma \left( \sum_{u \in N(v)} \alpha_{vu}^{(l)} \mathbf{W}^{(l-1)} \mathbf{h}_u^{(l-1)} \right) \quad (4)$$

The hidden layer of GCN is defined by Equation 3, starting with the input layer  $\mathbf{H}^{(0)} = \mathbf{X}$  and aiming for the output layer  $\mathbf{H}^{(L)} = \mathbf{Z}$ . The weight matrix, denoted as  $\mathbf{W}$ , is trainable. During the aggregate stage, the feature matrix  $\mathbf{H}$  is multiplied by the adjacency matrix  $\mathbf{A}$ . In the update stage, the

aggregated result is multiplied by the weight matrix  $\mathbf{W}$  and then passed through a non-linear activation function  $\sigma$ . In contrast, GAT introduces attention mechanisms to learn the relative weights between connected vertices. The graph convolutional operation in GAT is defined by Equation 4, where the attention weight  $\alpha_{vu}^{(l)}$  measures the connectivity strength between vertex  $v$  and its neighbor  $u$ .

### 3 GRAPH PRE-PROCESSING: TACKLING CHALLENGES IN GRAPH PROCESSING

In this section, we explore hardware-level challenges in graph processing execution. By exploring these challenges, we emphasize the crucial role of GPP methods in enhancing graph processing performance. We begin with a graph characteristics overview, detailing distinct execution behaviors that arise. We then analyze challenges stemming from these behaviors, emphasizing the significance of GPP in addressing them for efficient graph processing.

#### 3.1 Characteristics of Graph

Real-world graphs usually have the following three characters:

- **Irregularity:** Due to the inherent uncertainty of real-world scenarios, connections between objects in graphs are highly random, resulting in unstable numbers of links and variable linked objects for each vertex. Unlike regular grid structures or linear arrangements found in images or text, graph topology exhibits significant irregularity.
- **Power-law Distribution:** Real-world graphs often exhibit a power-law distribution in vertex degrees [39]. This means most vertices have relatively few neighbors, while a few vertices have a significant number of neighbors. For example, in social networks, a small number of celebrities may have a substantial number of followers, while the majority of ordinary users have considerably fewer followers.
- **Large Scale:** Real-world graphs are often characterized by their immense size. As of 2023, Facebook<sup>2</sup> boasts 2.25 billion vertices representing active users, while Twitter has 372.9 million vertices dedicated to active users. Furthermore, the World Wide Web<sup>3</sup> encompasses a staggering minimum of 6.19 billion vertices, representing pages, along with hundreds of billions of edges to signify links. Additionally, real graphs frequently involve feature vectors with dimensions exceeding thousands, further amplifying the overall scale of the graph.

#### 3.2 Execution Behavior of Graph Processing

The execution behavior of graph processing algorithm is profoundly influenced by the characteristics of the graph data, data storage format and the algorithm model. Figure 5 visually compares BFS and GCN execution. To comprehensively analyze the execution behavior of graph processing, we focus on four aspects: computing mode, computing intensity, memory access mode, and data reuse rate. These aspects significantly impact algorithm efficiency and performance. Table 3 provides an overview of the execution behavior of graph processing algorithms.

Table 3. Execution behaviors of graph processing

Behavior	Traditional Graph Computing (TGC)	Graph Neural Network (GNN)	
		Aggregate	Update
Computing Mode	Dynamic & Irregular	Dynamic & Irregular	Static & Regular
Computing Intensity	Low	Low	High
Memory Access Mode	Indirect & Irregular	Indirect & Irregular	Direct & Regular
Data Reuse Rate	Low	Low	High

<sup>2</sup><https://datareportal.com/essential-facebook-stats>

<sup>3</sup><https://www.worldwidewebsize.com/index.php?lang=NL>

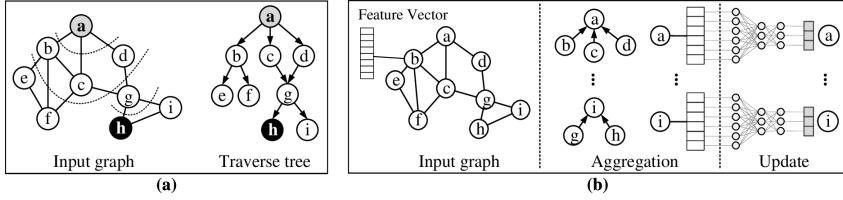


Fig. 5. Examples of the execution process of TGC and GNN: (a) Traversal process of BFS; (b) Each iteration of GCN first performs an aggregate operation to gather vertex information and then performs a combine operation for NN transformation.

- **Computing Mode:** TGC often involves traversing the neighbors of each vertex, a process also present in the aggregate stage of GNN. During traversal, information is passed between neighboring vertices, and the computation may vary depending on the graph structure and vertex attributes, resulting in a dynamic computing pattern. Additionally, the irregularity of the graph topology leads to a random number of neighbors being visited each time, creating an irregular computing pattern. On the other hand, the update stage processes the vertex attributes of entire graph at once, applying NN transformations to generate the output vertex feature vectors. As there is no need to traverse the graph iteratively, the computation is independent of input data, which means the computation is static. Furthermore, different vertices share the same neurons, resulting in a regular computing pattern in this stage.

- **Computing Intensity:** Computing intensity gauges the proportion of computation against data accessed from memory, which is a vital metric that reveals performance bottlenecks. In TGC, the vertex attributes typically comprise only a single element, yielding relatively simple operations like accumulation or comparison. As a result, the computing intensity is low and the performance is more constrained by memory access operations. Similarly, the aggregate phase in GNNs involves traversal operations to aggregate neighbors, exhibiting low computing intensity. Conversely, the update stage involves NN transformations to update high-dimensional vertex feature vectors for all vertices, exhibiting high computing intensity. So the performance in update stage is chiefly constrained by the computation itself.

- **Memory Access Mode:** Graph data is commonly stored in a compressed format. During graph traversal, the attribute data of neighboring vertices is accessed using the neighbor vertex numbers as indices, resulting in indirect memory access. The irregularity of the graph data leads to discontinuous addresses in this indirect memory access, resulting in irregular memory access patterns. In contrast, during the GNN update stage, all vertices undergo updates, and the high-dimensional vertex feature vectors are sequentially accessed directly from contiguous memory locations. Therefore, the GNN update stage exhibits a direct and regular memory access pattern.

- **Data Reuse Rate:** The reuse rate is a critical factor that affects computing efficiency and scalability during execution. It refers to the ability to reuse intermediate data, including intermediate results and cached input data. However, the irregularity of graph structures often leads to poor data locality and low data reuse rates in graph traversal processes between iterations. In contrast, the update stage of GNN demonstrates higher data reusability, as different vertices can share the same NN transformations and weights, leading to more efficient data utilization.

### 3.3 Challenges of Graph Processing

Graph processing algorithms encounter various challenges due to their distinctive execution behavior. In this section, we will analyze these challenges from a hardware perspective, specifically considering computing, storage, and communication aspects. To provide a visual overview, Figure 6 depicts the general hardware hierarchy and summarizes the specific challenges faced by graph processing at the hardware level.

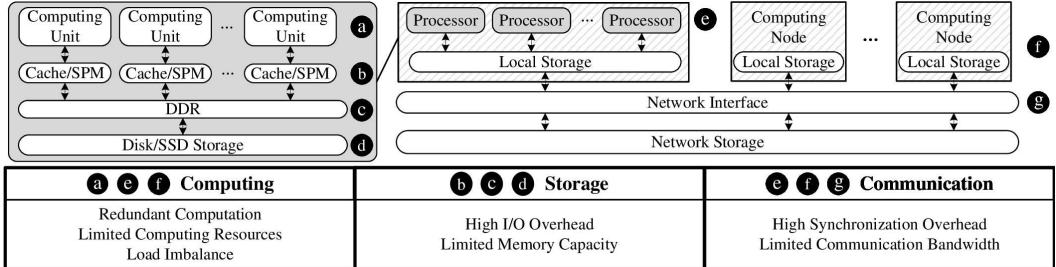


Fig. 6. Hardware hierarchy and hardware-level challenges of graph processing: **a b c d** together form a processor **e**, while **f** represents a machine housing multiple processors. Also, **f** signifies a computing node within a distributed system. The computing components of a parallel system encompass individual processors within a single machine and computing nodes within a distributed system. Typically, a machine housing multiple CPUs is referred to as a multi-CPU system, and a distributed system with multiple CPUs is referred to as a CPU-clusters system.

#### • Challenges in Computing **a e f**

*Redundant Computation:* This occurs when the same information is repeatedly calculated during algorithm execution, leading to low utilization of computing resources and longer computing times, significantly impacting the performance and scalability of graph processing systems. The irregularity of the graph can cause multiple visits to the same vertex during traversal, resulting in redundant computations. Moreover, low data reusability during traversal means intermediate results are not effectively stored, requiring the recalculation of the same data. In parallel systems, different cs may redundantly process the same data when the graph is shared among multiple computing components.

*Limited Computing Resources:* The problem arises when the size and complexity of the graph surpass the available computing power. Real-world graphs are often large, requiring massive computing power for graph algorithm execution. Additionally, some algorithms involve multiple iterations or recursive calculations, demanding a substantial amount of computing resources. For GNNs, the update stage involves computationally intensive calculations of high-dimensional feature vectors, potentially causing computing overload.

*Load Imbalance:* In large-scale graph processing, workload distribution across multiple components in a parallel system may become uneven. The irregular structure of the graph makes it challenging to partition vertices and edges into equal-sized subgraphs, leading to load imbalance. Some computing components become heavily loaded while others are underutilized, resulting in potential performance degradation. Addressing load imbalance is crucial for efficient and scalable graph processing.

#### • Challenges in Storage **b c d**

*High I/O Overhead:* The I/O overhead in graph processing refers to the time and resources consumed during the movement of data between different storage layers. This overhead is mainly attributed to irregular data access patterns. Due to the irregularity of graphs, data is often not stored contiguously in memory, leading to a large number of random I/O operations, which are significantly slower than sequential access and result in high storage access overhead. The presence of high I/O overhead in graph processing can significantly impact overall performance and scalability of graph algorithms, leading to longer execution times and hindering the ability to efficiently process large graphs.

*Limited Memory Capacity:* Large-scale graphs can exceed the available memory capacity, posing challenges to execution efficiency. Graph processing algorithms typically load the entire graph data into memory for efficient calculations. However, when the graph size surpasses the available

memory, some parts of the graph must be swapped in and out of memory, leading to data overflow. This situation causes numerous off-chip memory accesses, resulting in additional delays and substantial performance degradation.

- **Challenges in Communication** [e](#) [f](#) [g](#)

*High Synchronization Overhead:* Parallel graph processing requires coordination and synchronization among multiple computing components. As graph processing algorithms often depend on information from other components, frequent synchronization is necessary to ensure consistent computing results, resulting in high synchronization overhead. Moreover, inadequate local cache on components may lead to ineffective storage of shared data, resulting in repeated requests for the same data from other computing components, adding to the communication overhead. Excessive synchronization can cause communication delays and reduce the overall system execution efficiency.

*Limited Communication Bandwidth:* This is another crucial problem in parallel graph processing, referring to situations where the available network bandwidth for communication between computing components is insufficient. In large-scale GNNs, high-dimensional feature vector data often demands significant data transmission, putting substantial pressure on the network bandwidth. Additionally, the irregular execution behavior of graph algorithms leads to unpredictable communication patterns, making it challenging to efficiently utilize the communication bandwidth.

### 3.4 The Significance of Graph Pre-processing (GPP)

To address the challenges in graph processing, various graph processing frameworks and custom accelerators have been proposed. However, one of key factors in achieving efficient execution of graph algorithms is the pre-processing of graph data. GPP plays a vital role in optimizing the graph data and preparing it for efficient algorithm execution, thereby significantly improving the overall performance and scalability of graph processing systems.

Notably, the majority of graph processing work relies on GPP methods to tackle the complexities of large-scale graphs. A crucial step in processing such graphs is partition, which aims to achieve load balance and reduce communication overhead. Parallel frameworks like Giraph [6], and AliGraph [112] extensively utilize graph partition, as it significantly influences the data distribution across multiple computing components, enabling efficient parallel processing.

Moreover, GPP techniques are adopted to alleviate memory pressure and address the challenge of insufficient computing resources caused by large graphs. A widely adopted technique is the sampling-based mini-batch GNN training. GraphSage [43] is a classic sampling model used in various works, including GNN accelerators like HyGCN [108], AWB-GCN [34], and GNN training frameworks like NeuGraph [72], BGL [64], etc. Reorganization methods, such as those used in frameworks like CuSha [54], GridGraph [137], and custom accelerators like FPGP [24], HyGCN [108], etc., are also commonly employed to improve data access flow and further enhance the graph processing efficiency.

Furthermore, GPP becomes even more critical with the increase of heterogeneous platforms. GPP such as reorder and reconstruction, are usually performed on CPUs to prepare the graph data for acceleration on custom platforms. For example, GraphACT [118] maximizes CPU and FPGA utilization, further optimizing the overall performance.

Additionally, technologies in NNs, such as sparsification and quantization, are widely extended to GNNs to reduce the amount of data and improve computational efficiency, typical works include DropEdge [84], BiFeat [73], etc.

In conclusion, effective GPP techniques are crucial in enhancing the performance, scalability, and efficiency of graph processing. They enable the optimization of graph data, distribution across computing components, and preparation of specialized accelerators, all of which play a crucial role

in facilitating efficient graph processing. In the following sections, we propose our taxonomy and delve into a detailed analysis of existing GPP methods based on this taxonomy.

#### 4 GRAPH PRE-PROCESSING: TAXONOMY WITH DOUBLE-LEVEL DECISION

In this section, we present a comprehensive GPP methods taxonomy, utilizing a double-level decision framework, as shown in Figure 7. *In algorithmic perspective*, we categorize the seven methods into graph representation optimization and data representation optimization based on the optimization factors. *In hardware perspective*, we analyze GPP effects, classifying GPP methods into efficient computing, storage, and communication. This framework enhances understanding. Next, we outline our taxonomy and explain the rationale behind our classification.

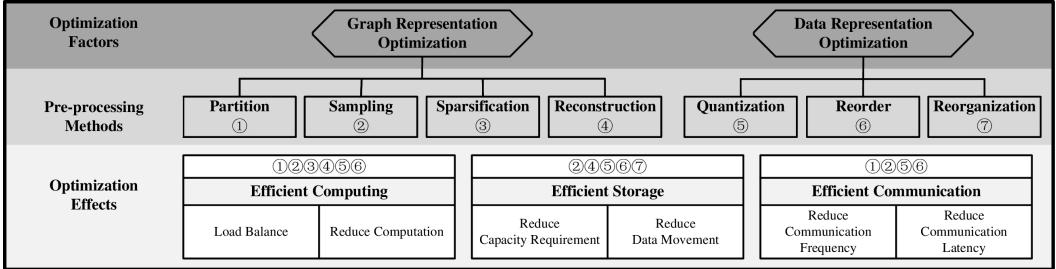


Fig. 7. Taxonomy of GPP methods: The GPP methods are classified using a double-level decision framework. The first level considers optimization factors from an algorithmic perspective, while the second level examines optimization effects from a hardware perspective.

##### 4.1 Taxonomy in Algorithmic Perspective

GPP entails two types of input graph data adjustments: graph representation optimization and data representation optimization. Graph representation optimization enhances graph algorithm performance by altering graph topology or density, while data representation optimization adjusts storage order or compresses data precision. By employing these two optimizations, researchers can explore various GPP approaches to improve the efficiency and effectiveness of graph algorithm execution while striking a balance between algorithm performance and accuracy.

- **Graph Representation Optimization:** This group encompasses partition, sampling, sparsification, and reconstruction techniques. Challenges in executing graph processing algorithms often arise from the irregularity of graph structures. These GPP methods modify input graph structures to enhance memory access and algorithm execution efficiency. Notably, partition, sampling, and sparsification may result in information loss, which could potentially affect the accuracy. In contrast, reconstruction solely alters the topology without impacting the algorithm's final outcomes.

- **Data Representation Optimization:** This group includes quantization, reorder, and reorganization techniques. Unlike graph representation optimization methods, these techniques do not alter the graph topology but instead focus on adjusting data storage. However, quantization reduces the precision of the data, introducing a trade-off between accuracy and execution efficiency. On the other hand, reorder and reorganization primarily adjust the data access mode to improve performance without affecting the algorithm's final outcome.

##### 4.2 Taxonomy in Hardware Perspective

Considering the challenges highlighted in Section 3 and the optimization goals, GPP offers opportunities to optimize graph algorithm execution in three aspects: efficient computing, efficient storage, and efficient communication. By effectively managing computing workloads, optimizing

memory access, and minimizing communication overhead, GPP significantly improves the performance and resource utilization of graph processing systems.

- **Efficient Computing:** Improving computing efficiency can be achieved from two perspectives: load balance and computation reduction. Firstly, load balance ensures an even distribution of computational work among resources, maximizing their utilization and avoiding overloading computing resources. It aims to achieve efficient utilization of available computing units, leading to improved performance. Secondly, reducing computation overhead can be achieved by minimizing the computation amount, including the reduction of redundant computation and data volume. GPP methods for efficient computing include partition, sampling, sparsification, reconstruction, quantization and reorder.

- **Efficient Storage:** It can be achieved by reducing capacity requirements and reducing data movement. Firstly, to reduce capacity requirements, it involves minimizing the amount of data buffered on-chip, consequently reducing I/O overhead. An effective approach is to reduce the overall data volume. Secondly, minimizing data movement can significantly improve memory access bandwidth utilization. By effectively managing storage resources and optimizing data movement, the storage efficiency can be enhanced, then improving overall performance and resource utilization. GPP methods for efficient storage include sampling, reconstruction, quantization, reorder, and reorganization.

- **Efficient Communication:** It can be benefited from reducing communication frequency or latency. Firstly, by reducing communication frequency, the synchronization overhead of computing components can be minimized, including processors in a single machine and computing nodes in a distributed system. Improving data locality is a useful way to reduce data exchange needs between components. Secondly, reducing communication delay means making full use of communication bandwidth. One effective method is to reduce irregular and redundant communication requests. GPP methods for efficient communication include partition, sampling, quantization, and reorder.

## 5 GRAPH PRE-PROCESSING IN ALGORITHMIC PERSPECTIVE

In this section, we will provide an overview of seven algorithmic GPP methods that are categorized into two distinct groups: graph representation optimization and data representation optimization. Our aim is to provide a comprehensive understanding of these methods by delving into their fundamental principles and exemplifying their typical algorithms. By the end of this section, readers will gain insights into the key concepts and practical implementation of these GPP methods. Typical algorithms of these GPP methods are listed in Table 4.

### 5.1 GPP Methods for Graph Representation Optimization

① **Partition.** Graph partition is a critical process in parallel graph processing systems, serving as the pre-processing step in both TGC and GNN. It has two primary objectives. Firstly, it evenly distributes vertices and edges across different components to achieve load balance, which ensures an efficient utilization of computing resources. Secondly, it focuses on minimizing split edges to improve data locality, thereby reducing communication overhead between computing components. As a result of pursuing these two objectives simultaneously, graph partition is recognized as an NP-hard problem. The general expression for the graph partitioning process is as follows:

$$\text{Partition}(V) = \{\bigcup_i V_i | i = 1, \dots, k; \forall i \neq j, V_i \cap V_j = \emptyset, \} \quad (5)$$

$$\{SubVertexSet_i | i = 1, \dots, k\} = \text{LoadBalance}(V_1, \dots, V_k) \quad (6)$$

$$\{Subgraph_i | i = 1, \dots, k\} = \text{MinEdgeCut}(E, \{SubVertexSet_i | i = 1, \dots, k\}) \quad (7)$$

Graph partition methods can be classified into two types [12] based on the cutting object: edge-cut partition and vertex-cut partition. In edge-cut partition, vertices are assigned to different subgraphs, resulting in edge-cuts that split edges crossing different subgraphs. In vertex-cut partition, edges are assigned to different subgraphs, and vertices may be replicated across multiple subgraphs. Partition can be performed offline, where the entire graph is divided in advance. For large-scale graph processing, streaming partition methods are used, continuously reading in edges or vertices for real-time partition. These online methods have lower time complexity and handle dynamic graphs efficiently. However, online partition may be less effective for lacking complete graphs. Next, we introduce some typical partition algorithms.

Table 4. GPP methods in algorithmic perspective

Method	Works
Partition ①	Edge-cut: RandomHash [74], METIS [52], LDG[91], Fennel [96], Chunk-based[136] Vertex-cut: GreedyEdge [39], DBH [104], HDRF [82], NE [123]
Sampling ②	Node-wise: GraphSAGE [43], PinSage [114], SSE [27], VR-GCN [113] Layer-wise: FastGCN [17], LADIES[139], AS-GCN [116] Subgraph-Based: GraphSAINT [119], Cluster-GCN [23], RWT [9]
Sparsification ③	Accuracy-oriented: DropEdge [84], FastGAT [90], NeuralSparse [131] Execution-oriented: GLT [19], Dynamic Pruning [15], AdaptiveGCN [58]
Reconstruction ④	Neighbor-merging: GraphACT [118] Neighbor-caching: HAG [49]
Quantization ⑤	Degree-free: EXACT [69], Bi-GCN [99], DGCNN [8] Degree-based: DegreeQuant[93], SGQuant [32], DBQ[41]
Reorder ⑥	Matrix-based: RCM [29], ParallelRCM[51] Degree-based: Hub-Sorting [129], Hub-Clustering [10], DBG [30], GNNTiering [78] Community-based: NestedDissection [57], LSH [20], Gorder [102], Recall [56], RabbitOrder [4]
Reorganization ⑦	Shard: Interval-and-Shard [24, 55, 85, 108], G-Shard [54] Chunk-Block: GridGraph [25, 137], NXGraph [22], NeuGraph [72]

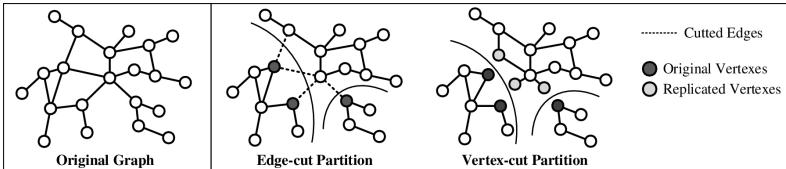


Fig. 8. Graph partition methods: Edge-cut partition vs. vertex-cut partition

- *Edge-cut Partition*

*RandomHash* [74] method is a simple heuristic approach commonly used as a baseline for initial graph partition experiments. It utilizes a random hashing function to partition the entire graph data. Although easy to implement and beneficial for small to medium-sized graphs due to its load balance and minimal storage requirements, it may not maintain locality within subgraphs for larger or more complex partition tasks, resulting in increased cut edges.

*METIS* [52], a popular open-source multilevel graph partition method, follows three steps: coarsening, partitioning, and uncoarsening. Coarsening reduces the graph by merging vertices into supernodes; partitioning allocates vertices across partitions; uncoarsening projects the partitioning back to the original graph. However, due to the need to traverse the entire graph, METIS consumes significant memory and is less efficient for large-scale graphs.

*LDG (Label Driven Greedy)* [91] is a widely used streaming graph partition algorithm. It follows a multilevel approach, employing a label-driven greedy strategy and thinning to achieve balanced

partitions with reduced edge cuts. This algorithm is highly efficient and effective in handling large-scale graphs with irregular structures.

*Fennel* [96] is a streaming graph partition algorithm that follows a similar multilevel approach with LDG. It stands out in its initial partitioning stage, leveraging spectral partition techniques, offering better communication efficiency.

*Chunk-based* partition [136] divides vertices into chunks and assigns them to cluster nodes. This method aims to achieve load balance by dynamically reassigning graph chunks, thus facilitating efficient parallel computing.

- *Vertex-cut Partition*

*GreedyEdge* [39] is a heuristic streaming graph partition method, which often serves as a practical baseline. It iteratively assigns edges to partitions based on a greedy strategy. Though it offers simplicity for large-scale graph partition, it may not always achieve the globally optimal partition and can be sensitive to the initial partition.

*DBH* [104] is a streaming graph partition method that utilizes a dynamic hashing function to assign vertices to partitions. It prioritizes dividing high-degree vertices to prevent splitting low-degree vertex communities and minimize communication during graph updates. DBH is memory-efficient but requires exploring the degree of each vertex.

*HDRF* [82] is a streaming partition method that combines hierarchical partition and degree-based randomized fusion. The hierarchical partition recursively divides the graph into smaller subgraphs, facilitating large-scale graph handling. Degree-based randomized fusion groups highly connected vertices to improve communication efficiency.

*NE* [123] is a heuristic method that considers the locality of neighbors to minimize vertex-cut.

② **Sampling.** In GNN training, employing a full-batch strategy introduces two key issues: memory constraints when traversing large graphs and slow convergence due to infrequent updates. Mini-batch training offers a solution by loading a limited number of vertices in each iteration, relieving memory pressure and accelerating convergence. Sampling selects a vertex subset for mini-batch creation, ensuring efficient training and scalability for large graphs. The sampling-based training addresses challenges in full-batch training, providing improved memory usage and faster convergence.

The sampling process in GNNs can be either online or offline. In offline sampling, the GNN model uses a fixed set of samples (i.e., subgraphs) from the graph data throughout the entire training or inference process, reducing computing and memory overhead. This approach is suitable when the graph data remains static during training. Conversely, in online sampling, the GNN model dynamically selects new samples from the graph data at each iteration during training. This adaptability allows the model to handle large-scale graphs that cannot fit entirely in memory and is particularly useful for processing dynamic graphs or streaming data. The graph sampling process is formulated as follows:

$$SN(v) = \text{Sampling}^{(l)}(N(v)) \quad (8)$$

$$\mathbf{h}_{N(v)}^{(l)} = \text{Aggregate}^{(l)}(\{\mathbf{h}_u^{(l-1)} : u \in SN(v)\}) \quad (9)$$

$$\mathbf{h}_v^{(l)} = \text{Update}^{(l)}(\{\mathbf{h}_v^{(l-1)}, \mathbf{h}_{N(v)}^{(l)}\}) \quad (10)$$

Graph sampling methods can be classified into three types: node-wise sampling, layer-wise sampling, and subgraph-based sampling [66]. Node-wise sampling selects vertex neighbors as subgraphs, but the total neighbors grow exponentially with increasing layers. Layer-wise sampling overcomes this by sampling a fixed number of vertices in each layer, but it may lead to sparser subgraphs in deeper layers. Subgraph-based sampling samples vertex and edge sets starting from an initial vertex, providing better independence. Next, we introduce some typical sampling models.

- *Node-wise Sampling*

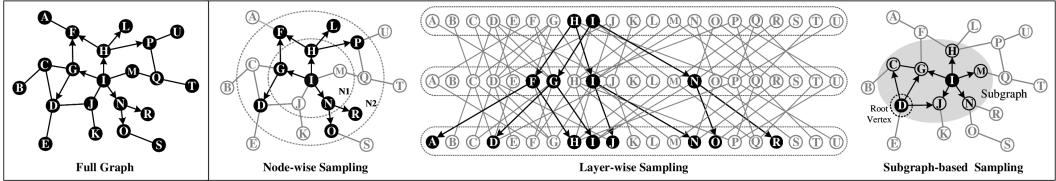


Fig. 9. Graph sampling methods: Node-wise sampling vs. layer-wise sampling vs. subgraph-based sampling

*GraphSAGE* [43] is a popular graph representation learning algorithm. It leverages local neighborhood information of vertices in a graph to perform inductive learning and learn node embeddings. During the training process, GraphSAGE samples a fixed-size neighborhood for each vertex, enabling efficient scalability to large graphs. While GraphSAGE primarily operates in an offline manner, it can also be combined with online sampling techniques for specific needs.

*PinSAGE* [114] is an extension of GraphSAGE designed for personalized recommendations. It utilizes random walk for expansion and combines GraphSAGE neighborhood sampling and aggregation techniques with personalized information to enhance accuracy in personalized scenarios. Like GraphSAGE, PinSAGE is mainly used offline.

*SSE* [27] is an online sampling method used in large-scale graph processing. It performs random neighbor sampling for only 1-hop neighbors, avoiding the exponential growth of neighbors. SSE efficiently samples edges from the graph in real-time during processing, reducing memory requirements, and making it more scalable for processing large graphs.

- *Layer-wise Sampling*

*FastGCN* [17] is an efficient and scalable graph sampling method proposed to accelerate GCN training. It uses a two-step strategy to select fixed-size neighborhoods for each vertex, improving computing efficiency. While primarily offline, it can be adapted for online sampling if needed.

*LADIES* [139] is an online graph sampling method. It reduces computation overhead by discarding unnecessary vertices and edges during the sampling process, using a layer-wise discarding strategy for enhanced sampling efficiency.

*AS-GCN* [116] is an online adaptive sampling method, selecting informative samples based on the importance. It samples a fixed number of vertices in each layer in a top-down manner, reusing vertices to improve training efficiency.

- *Subgraph-based Sampling*

*GraphSAINT* [119] is an offline sampling method that estimates the probabilities of vertex and edge sampling separately. It aims to select informative and diverse subgraphs, thereby enhancing the training efficiency of GCNs.

*Cluster-GCN* [23] is another offline graph sampling method that efficiently handles large graphs by employing clustering and mini-batch sampling. It significantly reduces memory usage during training, particularly for deep GCNs.

*RWT* [9] is an offline sampling method for training GCNs that utilizes random-walk sampling to construct mini-batches. It aims to sample diverse vertex neighborhoods, thereby enhancing the generalization capabilities of GCNs.

**③ Sparsification.** GNNs use sparsification to enhance training and inference efficiency. This technique selectively removes unnecessary edges, reducing redundant computations and storage burdens. Additionally, we consider dynamic pruning as a sparsification method in this survey, as it creates a more compact graph for execution.

The sparsification process can be executed either online or offline, providing versatility for various applications. For graphs that evolve over time, sparsification efficiently removes redundant information and adapts to changing graph structures. In contrast, in static graphs, sparsification is

valuable for reducing memory and computing overhead, especially for large graphs. The process of sparsification is shown in Figure 10 (a), and it can be formulated as follows:

$$\mathbf{A}_{sp} = \text{Sparse}(\mathbf{A}) = \text{RemoveEdge}(\mathbf{A}) \rightarrow \text{Training or Inference GNN}(\mathbf{A}_{sp}) \quad (11)$$

Sparsification methods fall into two categories based on their purpose: accuracy-oriented and execution-oriented. Accuracy-oriented sparsification aims to enhance training accuracy by eliminating redundancies. However, it may lead to increased runtime due to more iterations. On the other hand, execution-oriented sparsification focuses on accelerating GNN execution and reducing computation and storage burdens. Next, we introduce some typical algorithms.

- *Accuracy-oriented Sparsification*

*DropEdge* [84] is an offline method which randomly drops edges during training, sparsifying the graph and improving model robustness and generalization. It reduces computing complexity and memory requirements for deep GCNs.

*FastGAT* [90] is an offline method using effective resistance-based graph sparsification to reduce computing complexity while maintaining performance in GAT training.

*NeuralSparse* [131] is an offline method guided by a NN that prunes edges with low importance scores.

- *Execution-oriented Sparsification*

*GLT* [19] generalizes the lottery ticket hypothesis from NNs to GNNs. By identifying a sparse subnetwork, GLT achieves comparable performance to the original dense network but with a smaller number of parameters. It is considered offline since it involves pre-training the dense network and then pruning it to find the winning ticket.

*AdaptiveGCN* [58] introduces an adaptive sparsification technique for GCNs. It dynamically sparsifies the graph during training by adaptively removing edges based on their importance. This method can be categorized as online since it involves dynamic pruning of edges during training.

*Dynamic Pruning* [15] performs edge pruning during the training process to create a more compact and efficient graph representation. As it involves dynamic pruning during the training phase, it is considered as an online sparsification.

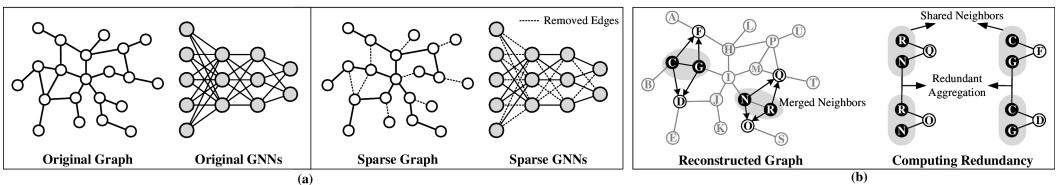


Fig. 10. An illustration of sparsification and reconstruction process: (a) Graph sparsification through removing edges; (b) Graph reconstruction through merging shared neighbors

④ **Reconstruction.** GNNs employ NN-based neighborhood message passing mechanisms to update target vertex representations by aggregating feature messages from neighboring source vertices. However, in complex graphs, a common scenario is the presence of multiple shared neighbors, leading to redundant calculations as these shared neighbors are aggregated repeatedly. To overcome this challenge, a technique called "reconstruction" is introduced. As shown in Figure 10 (b), this approach enables the reuse of intermediate aggregation results, thus enhancing the efficiency of computing and storage.

The reconstruction process causes no information loss since edges are not discarded. Reconstruction can be performed online or offline, depending on the applied scenarios. By adopting reconstruction, GNNs gain the promotion in terms of efficiency without compromising the accuracy of the graph data. The reconstruction technology can be categorized into two groups: neighbor-merging and neighbor-caching. Next, we introduce typical works in each groups.

- *Neighbor-merging Reconstruction*

GraphACT [118] merges shared neighbor pairs before training to effectively reduce redundant neighbor reduction operations. Since the merging occurs per epoch, it is performed online. Through data reconstruction, the data reuse rate is increased, resulting in improved execution efficiency.

- *Neighbor-caching Reconstruction*

HAG [49] caches and reuses intermediate results for subsequent vertices with similar neighborhoods. The caching of intermediate results occurs during the pre-processing step of the GNN, where the GNN traverses the graph and computes the intermediate aggregation results for each vertex. These results are then stored in a cache or lookup table. This combination of offline caching and online reuse reduces unnecessary overhead.

## 5.2 GPP Methods for Data Representation Optimization

⑤ **Quantization.** Data quantization in GNNs involves converting continuous floating-point feature vectors into discrete values with a limited number of bits. This process reduces computational and memory overhead while sacrificing some model accuracy. Typically, high-dimensional feature data represented as 32-bit floating-point numbers is transformed into low-bit integer values, as shown in Figure 11 (a). Generalizing from NNs to GNNs, data quantization plays a crucial role in improving scalability and practicality on resource-constrained platforms.

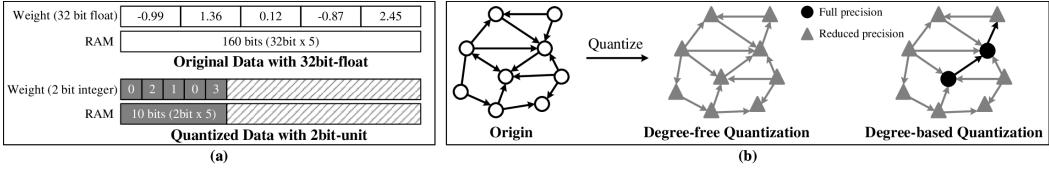


Fig. 11. An illustration of quantization: (a) The 32-bit floating-point weights are quantized into 2-bit integer data; (b) Diagram of degree-free quantization and degree-based quantization.

Graph quantization can be divided into two categories: degree-free and degree-based, as shown in Figure 11 (b). Degree-free quantization directly extends classical NNs quantization methods to GNNs without considering vertex degrees. However, due to the irregular graph topology, methods that account for vertex degree are more suitable. Degree-based quantization methods address the irregular topology by quantizing based on vertex degrees, aiming to minimize data scale with an acceptable accuracy loss. Next, we will introduce some typical works.

- *Degree-free Quantization*

*EXACT* [69] introduces extreme activation compression for quantizing GNNs. By compressing training activations to a limited set of extreme values, storage and computation needs are minimized, enabling efficient large-scale graph training. The method employs quantization-aware training to mitigate accuracy degradation due to quantization.

*Bi-GCN* [99] proposes an offline quantization method that converts the continuous floating-point weights of GNNs into binary values. The approach also involves an adaptive scaling mechanism to preserve model accuracy.

*DGCNN* [8] employs binary representations for vertex features and graph structures to achieve model compression. Vertex features are binary-coded, reducing memory overhead, while graph structures are encoded in binary forms.

- *Degree-based Quantization*

*DegreeQuant* [93] proposes a quantization-aware training technique for GNNs, considering the irregular topology by incorporating vertex degree. This reduces computation and memory overheads while preserving quantization accuracy.

*SGQuant* [32] focuses on squeezing the last bit to achieve efficient training and inference.

*DBQ* [41] identifies sensitive vertices in graph structures and applies a protective mask to ensure that these vertices perform full-precision calculations, while other vertices undergo quantization. This dynamic accuracy adjustment achieves greater acceleration without compromising classification accuracy.

⑥ **Reorder.** This method is a valuable data layout optimization technique that addresses the issue of poor memory locality in graph data caused by its structural irregularities. As depicted in Figure 12, by relabeling vertex IDs, reorder improves data locality, taking reduced irregular memory accesses and improved memory access bandwidth utilization. Reorder can be applied to both TGC and GNN, enhancing the efficiency and scalability of large-scale graph processing.

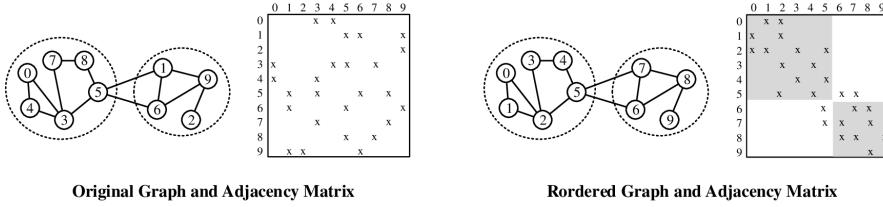


Fig. 12. Graph reorder process: The data locality is improved compared to the original scattered matrix with unmodified IDs.

Graph reorder techniques can be divided into three categories: matrix-based, degree-based, and community-based. Matrix-based reorder aims to compress the adjacency matrix bandwidth by reducing zero entries. While simple and fast, this approach may not fully consider the graph characters, leading to potential inefficiencies. In contrast, degree-based reorder takes vertex degrees into account. High-degree vertices are expected to be frequently accessed, and sorting them can improve cache utilization. Community-based reorder considers the power-law characteristics of the graph and groups closely connected vertices together, enhancing data locality. Next, we will describe some typical algorithms.

- *Matrix-base Reorder*

*RCM (Reverse Cuthill-McKee)* [29] is a graph reorder algorithm that reduces matrix bandwidth in sparse graphs. It aims to improve cache utilization and memory access patterns by minimizing the distance between non-zero entries. The algorithm selects a starting vertex and performs BFS to find neighbors, ordering them based on their degrees. Papers [7, 51] explore parallelization and distributed-memory implementation of RCM.

- *Degree-base Reorder*

*Hub-Sorting* [129] is a lightweight reorder technique that sorting high-degree vertices (hot vertices). It sorts hot vertices in descending order of degrees, placing them in a cacheline to enhance cache efficiency.

*Hub-Clustering* [10] reorder technique clusters hot vertices with high-degree and places them at the beginning of the reordered sequence, but it does not further sort interior of these hot vertices clusters. Compared with Hub-Sorting, Hub-Clustering reduces the fine-grained reorder time overhead, but the spatial locality is relatively poor.

*DBG (Degree-base Grouping)* [30] reorder adopts a coarse-grained sorting approach to preserve the graph structure while reducing the cache occupancy of hot vertices. It roughly divides the vertices into groups based on the degree and then arranges groups in descending order, while maintaining the original vertex order within each group.

*GNNTiering* [78] optimizes GNN training on GPUs using the Weighted Reverse Pagerank method. It stores high-scoring vertices with high outdegree in GPU memory, reducing large-scale data transmission over PCIe.

- *Community-base Reorder*

*NestedDissection* reorders and partitions graphs to reduce Cholesky factorization fill-in in sparse matrices. It recursively removes separator vertices and employs partition and reorder operations, with parallelization in mt-METIS [57].

*LSH* [20] is a typical reorder technique that utilizes locality-sensitive hashing to group similar vertices together.

*Gorder* [102] is a heavyweight graph reorder algorithm that introduces a priority queue-based graph reorder algorithm (GO-PQ) to maintain frequently accessed vertices in a cacheline for improved cache utilization. While *Gorder* provides significant acceleration for large-scale graphs, it comes with high time overhead. Building on *Gorder*, *ReCall* [56] introduces a cache data reuse metric profile, yielding a heuristic pH, albeit with similarly high time overhead.

*RabbitOrder* [4] is the first just-in-time parallel reorder algorithm, which introduces a hierarchical communities-based approach derived from real-world hierarchical community structures. By mapping these communities into hierarchical caches, *RabbitOrder* leverages low-latency cache levels for improved performance.

⑦ **Reorganization.** Large-scale graph processing faces challenges in caching all on-chip data, leading to substantial off-chip data movement. The irregular graph structure prompts random memory accesses, constraining performance. To tackle this, data reorganization enhances data flow by optimizing layout, bolstering reusability, and maximizing storage bandwidth. This technique splits the graph into slices, adjusting data order within each slice. Notably, reorganized slices are accessed sequentially, while partitioned subgraphs are processed concurrently. Next, we showcase typical works.

*Shard* technology, introduced in Graphchi [55], optimizes data access on CPUs within a single machine. It splits graph data into intervals and shards (Figure 13 (a)), organizing source vertices sequentially for efficient parallel sliding window-based data access. Shards find application in various works: X-stream offers a parallel implementation, CuSha tailors G-shards for GPUs, FPGP employs them on FPGAs, and HyGCN utilizes them for GNN inference [24, 54, 85, 108].

*Chunk-Block* technique, from GridGraph [137], divides the graph into grids. It splits vertices into P uniform chunks of fixed size and edges into PxP blocks (Figure 13 (b)), utilizing dual sliding windows during computation to substantially reduce I/O overhead. Similar techniques are also employed in NXGraph [22], ForeGraph [25] and NeuGraph [72].

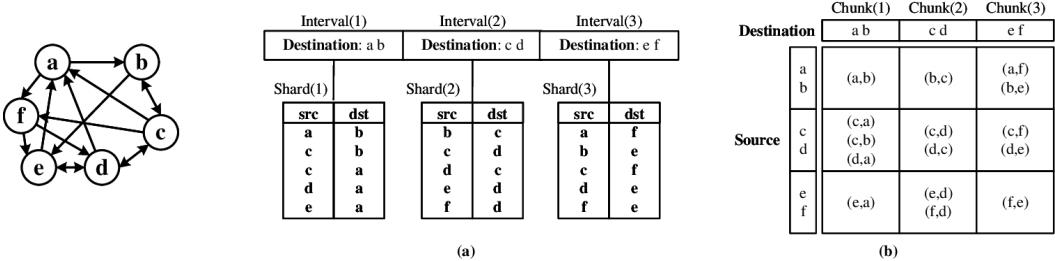


Fig. 13. Examples of data reorganization: (a) Interval-and-Shard in Graphchi [55]; (b) Grid representation in GridGraph [137]

## 6 GRAPH PRE-PROCESSING IN HARDWARE PERSPECTIVE

In the previous section, we provided an algorithmic perspective by introducing seven GPP methods. Now, our attention shifts to the hardware perspective, where we delve deeper into the optimization effects of these GPP methods. This section will be organized into three categories: *efficient computing*, *efficient storage*, and *efficient communication*. We will explore practical implementations and

examine related works that utilize these GPP methods to achieve optimization in each category. Our aim is to offer a comprehensive understanding of the hardware optimization possibilities that GPP methods present. Through the exploration of these related works, we will illustrate practical implementations and showcase the advancements made in this field.

## 6.1 GPP Methods for Efficient Computing

Table 5. Works with GPP methods for efficient computing

Effect	Method	Algo.	Works
Load Balance	Partition	TGC	Pregel [74], Giraph [6], Mizan [53], PowerGraph [39], GraphLab [70], Gemini [136], PowerLyra [18]
		GNN	AliGraph [112], DistDGL [132], PaGraph [63], Dorylus [94], QGTC [101]
Reduce Computation	Sampling	GNN	AliGraph [112], DistDGL [132], AGL [124], PaGraph [63], 2PGraph [125], P3 [33], DGTP [71], Pasca [128], QGTC [101], Sugar [106]
	Sparsification	GNN	GLT [19], DyGNN [15]
	Reconstruction	GNN	GraphACT [118], GCNInfer [120], HAG [49], I-GCN [35], ReGNN [14]
	Quantization	GNN	FPGAN [111], Bi-GCN [99], QGTC [101]
	Reorder	TGC	Sparc [134], GraphA [36]

**6.1.1 Load Balance.** To achieve load balance in parallel graph processing systems with multiple computing components, graph partition is commonly utilized to evenly distribute vertices and edges among different components. Next, we will explore relevant works in graph partition.

- **Partition.** Many parallel graph processing works utilize graph partition to achieve load balance, applicable to both TGC and GNN algorithms. Notably, a considerable number of parallel GNN training works employ a sampling-based strategy, thereby necessitating a thoughtful integration of graph partition with subsequent graph sampling.

*Pregel* [74], developed by Google, is a distributed TGC framework that operates within the batch synchronous parallel (BSP) model. It employs static graph partition to ensure an even distribution of computing tasks across computing nodes for load balance. *Giraph* [6] extends Pregel's capabilities by integrating it with Hadoop.

*Mizan* [53] introduces dynamic partition to Pregel for dynamic workloads. Mizan dynamically adjusts partition boundaries, promoting dynamic load balance by redistributing workloads and maintaining vertex uniformity.

*PowerGraph* [39] is a parallel TGC system, optimizing power-law graph processing through a greedy edge-cut partition strategy. It minimizes synchronization by only focusing on edge-connected vertices, mitigating the imbalance caused by intensive computations on a single node.

*GraphLab* [70] is an open-source distributed TGC framework that employs a vertex-centric model and a variety of graph partition algorithms to guarantee a well-proportioned distribution of computational tasks.

*PowerLyra* [18] is an extension of GraphLab, achieving load balance by effectively distributing high-degree vertices across nodes. It introduces a customized hybrid partition strategy optimized for power-law graphs, utilizing an edge-cut partition for low-degree vertices and a vertex-cut partition for high-degree vertices.

*Gemini* [136] is a distributed TGC framework, improving load balance through chunk-based graph partition. It dynamically allocates vertices to computing nodes, ensuring a fair distribution of workloads for better performance.

*AliGraph* [112] is Alibaba’s distributed GNN training system that reduces edge crossings among threads via partition. The system offers a range of partition methods for selection and supports the addition of custom algorithms as plugins.

*DistDGL* [132] operates as a distributed GNN training framework, employing METIS for graph partition. It refines METIS by transforming its approach into a multi-constraint problem, leading to improved load balance.

*PaGraph* [63] supports GNN training using multiple GPUs within a single server, where each GPU is allocated an independent partition for training. It modifies the LDG partition method to suit graph sampling. To ensure even workload distribution across GPUs, every partition is equipped with a comparable count of training target vertices.

*Dorylus* [94] is a distinctive distributed system for GNN training that leverages serverless computing for scalable and cost-effective solutions. It employs chunk-based graph partition to achieve load balancing effectively.

*QGTC* [101] is a Tensor Core (TC)-based computing framework that can be used for GNN training and inference. It employs METIS for graph partition, distributing distinct subgraphs to multiple GPUs for efficient computation.

**6.1.2 Reduce Computation.** As described in Section 4, there are two primary strategies to minimize computation: eliminating redundant computation and decreasing data volume. Effective GPP methods for eliminating redundancy include sparsification, reconstruction and reorder. Meanwhile, data volume can be reduced through GPP techniques such as sampling and quantization. Next, we will explore relevant works using these GPP methods.

- **Sparsification.** By removing less influential or redundant edges, sparsification reduces the number of computations required during GNN operations, leading to faster training and inference times.

*UGS* [19] presents a unified sparsification framework for GNN training, concurrently pruning both adjacency matrix and model weights. By applying the GLT sparsification method, it focuses on the most impactful vertices and edges, significantly diminishing the computational complexity of GNNs. Consequently, it leads to substantial savings in multiply accumulate (MAC) computation during SPMM operations.

*DyGNN* [15] accelerates GNN inference through a collaborative sparsification algorithm and customized architecture co-design. It addresses early vertex convergence and redundant neighbor aggregation, dynamic pruning to reduce graph density while preserving structure, thereby reducing computation. DyGNN incorporates a specialized Pruner for on-the-fly pruning and a streamlined pipeline design for rapid, low-latency inference.

- **Reconstruction.** This method is employed in various GNN accelerators, effectively reducing aggregation redundancy. Some works use heterogeneous architectures, performing reconstruction on CPUs and graph processing on custom architectures. For further acceleration, more studies are customizing architectures for real-time reconstruction.

*GraphACT* [118] proposes a CPU-FPGA heterogeneous architecture to expedite GNN training. It executes real-time reconstruction operations on the CPU, scanning each mini-batch to identify and merge neighbor pairs proactively. Reconstruction computes the highly reusable portions of the graph in advance, effectively precluding the need for subsequent redundant neighbor vertex reduction operations, significantly alleviating the computational burden.

*GCNInfer* [120] is a customized GNN inference accelerator, employing a reconstruction approach similar to GraphACT. Through offline preprocessing, it merges common neighbors and removes redundant edge computations for high-degree vertices. Notably, in the inference phase,

which utilizes the entire graph, the effectiveness of redundancy elimination and computational enhancement surpasses that of the training phase with mini-batch subgraphs.

*HAG* [49] presents a novel hierarchical aggregation computation graph for managing intermediate results and reducing computation redundancy. Through online reconstruction, it caches intermediate aggregate results and utilizes them based on identified dependencies in subsequent iterations, effectively minimizing redundant computations during updates and enhancing overall computation speed for GNN training and inference.

*I-GCN* [35] is a GCN inference accelerator featuring an islandation module for online reconstruction. This module clusters vertices to identify those with shared neighbors. During aggregate phase, common neighbor aggregation data is reused, eliminating redundant computations. Due to its custom architecture, *I-GCN* is superior in reconstruction speed.

*ReGNN* [14] designs a GNN accelerator that harmonizes algorithm and hardware through dynamic redundancy elimination. It presents a novel approach involving a neighborhood message passing algorithm, which pre-aggregates shared neighbors and stores reusable intermediate results. This method is supported by a dedicated architecture, effectively transforming redundancy elimination into performance enhancement.

- **Reorder.** In ReRAM-based architectures, where computation occurs in memory, the presence of zero values in sparse matrices can result in considerable redundant computations. Reorder techniques enhance data locality, effectively mitigating sparsity and then improving overall energy efficiency.

*Sparc* [134] is a ReRAM-based TGC accelerator. It employs the ReRAM crossbar size as a threshold for vertex reorder, optimizing workload density within the ReRAM crossbar. Therefore, it achieves a high computing energy efficiency.

*GraphA* [36] is a TGC accelerator, comprising multiple ReRAM Graph Engines (RGEs). It assigns partitioned subgraphs to RGE units, then performs reorder based on the degrees, reducing sparsity to enhance computing resource utilization.

- **Sampling.** By selecting a small subset of vertices and edges for training, graph sampling decreases the vertex number involved in computing, ultimately reducing the overall computation complexity and facilitating faster iterations. Sampling technique is widely adopted, finding its application in various frameworks designed for GNN training.

Many distributed GNN training frameworks integrate partition with sampling. Some frameworks start by partitioning the large graph into subgraphs, which are then allocated to individual computing nodes for sampling. Such frameworks include AliGraph [112], DistDGL [132], PaGraph [63], 2PGraph [125], P3 [33], DistDGLv2 [133], Pasca [128], and Sugar [106]. In contrast, some frameworks directly sample the original graph and then distribute them to various computing nodes. Such frameworks include AGL [124] and DGTP [71]. The former one performs sampling within the assigned subgraph independently at each compute node, however, it may result in information loss, affecting convergence. The latter one samples the entire graph, preserving information integrity but consuming more resources for sampling.

- **Quantization.** By conversing high-precision floating-point data into low-precision integer data, quantization diminishes the overall workload, significantly curtailing the requirement for computing resources. Also, post quantization, matrix multiplication can be streamlined into simpler logical operations, reducing the computation complexity.

*FPGAN* [111] operates as an FPGA-based GAT inference accelerator. It quantizes input features, converting the original floating-point multiplication into a shift operation. Furthermore, its customized architecture enhances calculation speed while conserving computational resources.

*Bi-GCN* [99] introduces a binary GCN that binarizes both network parameters and vertex features. This conversion allows for the transformation of complex SPMM into straightforward binary operations, thus achieving acceleration.

*QGTC* [101], a TC-based GNN computing framework, proposes a novel quantized low-bit arithmetic design. This design utilizes low-bit data representation and bit-decomposed computation to convert complex high-precision floating-point matrix multiplication into faster bit-matrix multiplication, optimizing computing speed.

## 6.2 GPP Methods for Efficient Storage

Table 6. Works with GPP methods for efficient storage

Effect	Method	Algo.	Works
Reduce Capacity Requirement	Sampling	GNN	AliGraph [112], DistDGL [132], AGL [124], PaGraph [63], 2PGraph [125], P3 [33], DGTP [71], Pasca [128], QGTC [101], Sugar [106]
	Quantization	GNN	SGQuant [32], EXACT [69], DGCNN [8], BiFeat [73], DBQ [41]
Reduce Data Movement	Reconstruction	GNN	GraphACT [118], GCNInfer [120], HAG [49], I-GCN [35], ReGNN [14]
	Reorder	TGC	Cagra [129]
		GNN	GCNInfer [120], Rubik [20], H-GCN [121], GNNTiering [78]
	Reorganization	TGC	GraphChi [55], TurboGraph [44], X-Stream [85], CuSha [54], FPGP [24], GridGraph [137], NXGraph [22], ForeGraph [25], FPGA-Cache [5], Graphicionado [42], GraphDynS [109], GraphR [89], GraphSAR [26]
		GNN	NeuGraph [72], HyGCN [108]

**6.2.1 Reduce Capacity Requirement.** In graph processing, optimizing on-chip storage for initial data and intermediate results is crucial, since insufficient on-chip memory leads to slow off-chip I/O. A viable optimization strategy is to reduce the overall data volume, and two effective techniques for achieving this in GPP are sampling and quantization.

- **Sampling.** In contrast to full-batch GNN training, the sampling-based strategy only requires caching mini-batches, substantially reducing memory demands. As a result, this approach is more easily scalable and deployable, and it finds widespread usage in numerous studies. An overview of relevant distributed GNN training frameworks is presented in Table 6. Given that these frameworks have been introduced in Section 6.1.1, redundant explanations are omitted here.

- **Quantization.** By employing fewer bits to represent each value, quantization significantly reduces overall memory consumption. This reduction in memory requirements is particularly advantageous when on-chip memory is limited.

*SGQuant* [32] employs a multi-granularity quantization approach, utilizing layer-wise, component-wise, and topology-aware quantization granularity. It compresses GNN features while minimizing accuracy loss. Experiments show that *SGQuant* achieves a remarkable memory capacity reduction of up to 31.9 times while maintaining high training accuracy.

*EXACT* [69] is a GPU-based GNN training solution specifically designed for deep GNNs. Addressing the challenge of substantial activation storage requirements during training, *EXACT* employs compression techniques to markedly reduce memory overhead. Experimental results indicate that *EXACT* achieves memory savings of up to 32 times.

*DBQ* [41] introduces degree-based quantization, an efficient method to reduce memory usage. *DBQ* strategically quantifies vertices with minimal influence, striking a balance between precision loss and quantization compression ratio. This innovative approach effectively addresses memory constraints while maintaining a high level of accuracy.

*DGCNN* [8] stands out by utilizing binary representations for both vertex features and graph structures. This efficient approach involves the conversion of node features into binary codes, thereby reducing memory overhead. Moreover, the encoding of graph structures in binary format further contributes to computational efficiency and model compression.

*BiFeat* [73] puts forth a novel GNN feature quantization approach. This technique accommodates both scalar and vector quantization methods, effectively reducing the memory footprint of GNN feature data. While scalar quantization, based on the logarithmic method, offers quick GPP with modest compression, vector quantization achieves extensive data compression, making it suitable for large graphs albeit with more time-consuming compression calculations.

**6.2.2 Reduce Data Movement.** The irregularity of graph topology leads to inefficient and random memory access patterns, resulting in frequent data movement. Reorder and reorganization methods enhance dataflow access, thus improving memory bandwidth utilization. Additionally, the reconstruction technique consolidates redundant data, effectively reducing unwarranted transfers. Next we introduce works that leverages these GPP methods for optimization.

- **Reconstruction.** Reusing intermediate results through the reconstruction method effectively reduces redundant data movement. Notably, GraphACT [118], GCNInfer [120], and I-GCN [35] strategically decrease data transfer volumes by pre-merging shared vertices. Additionally, HAG [49] and ReGNN [14] enhance storage efficiency by caching reusable intermediate outcomes, effectively mitigating the need for repeated data reading.

- **Reorder.** This method significantly enhances data locality through the strategic rearrangement of graph vertices. This transformation effectively converts irregular memory access patterns into more regular sequences, thereby optimizing cache storage efficiency and mitigating cache misses.

*Cagra* [129] serves as a cache-optimized memory graph framework, designed to expedite TGC. This framework employs hub-sorting for vertex reorder. Cagra focuses on enhancing the gather phase of pull-based graph algorithms, strategically aggregating frequently accessed vertices based on out-degree to bolster cache efficiency.

*Rubik* [20] is a GCN inference accelerator that employs LSH reorder to enhance data reuse during aggregation.

*GCNInfer* [120] is a GCN inference accelerator that executes GCN using SPMM. To enhance data locality and maximize bandwidth utilization, it employs the RCM algorithm for vertex reorder.

*H-GCN* [121] stands as a hybrid accelerator utilizing PL (programmable logic) and AIE (AI engine) within the Xilinx Adaptive Computing Acceleration Platform for high-performance GNN inference. It utilizes open-source tool mt-metis to implement graph reorder, effectively clustering vertices with shared neighbors and optimizing data locality.

*GNNTiering* [78] is a multi-GPU GNN training framework that uses the GraphSage model for mini-batch training. It strategically improves storage access efficiency by employing statistical techniques to identify data locality. GNNTiering uses reorder method to allocate high-degree vertices to the GPU cache and routes low-degree vertices to the CPU. This minimizes PCIe transmission of high-dimensional feature vectors between the CPU and GPU.

- **Reorganization.** Large-scale graph processing often involves frequent data exchange between memory and off-chip storage due to memory limitations. Reorganization techniques offer a solution by optimizing data layout in advance, thereby enhancing data read and write efficiency while reducing the overhead associated with data movement.

*GraphChi* [55] is a prominent framework for efficient large-scale TGC. It introduces the "Interval-and-Shard" technique, partitioning graphs into intervals and shards through offline GPP, enhancing memory usage and computation by promoting data locality. GraphChi employs the Parallel

Sliding Window (PSW) method to sequentially access intervals and shards, minimizing random disk access, thereby improving storage bandwidth utilization and achieving high performance on a single machine. *TurboGraph* [44] and *X-stream* [85] present parallel improvements for GraphChi. This technique is also used in TGC accelerators Graphicionado [42] and GraphDynS [109].

*CuSha* [54] is a CUDA-based framework, accelerating TGC on a single machine with multiple GPUs. It introduces G-Shard and Concatenated Windows (CW) to harness GPU parallelism by splitting edges into shards and sequentially accessing storage using CW. This optimization ensures coalesced memory access, reducing data flow irregularities.

*FPGP* [24] stands as a FPGA-based streaming framework tailored for TGC. Similar to GraphChi, FPGP also adopts the Interval-and-Shard methodology. In this scheme, the graph's vertices are divided into  $P$  intervals, while the edges are partitioned into  $P^2$  sub-shards. Notably, updates are executed in sub-shard units. By leveraging Interval-and-Shard technology, FPGP optimizes storage access bandwidth utilization and achieves streaming execution of TGC algorithms.

*GridGraph* [137] enables large-scale TGC on a single machine through 2D-partitioning and dual sliding windows (DSW). It splits graph into vertex chunks and edge blocks, utilizing DSW to streamline edge streaming and dynamic vertex updates, reducing storage I/O overhead. Notably, the GPP costs in GridGraph is lower than that in GraphChi.

*NXGraph* [22], a single-machine TGC framework, employs a reorganization technique similar to GridGraph. Notably, it arranges destination vertices in descending order to enable sequential storage of edges with the same destination.

*GraphR* [89] is the pioneering work that leverages ReRAM to expedite TGC. It employs a method akin to GridGraph, dividing blocks and generating ordered edge lists to enhance the flow of data access. A similar work is *GraphSAR* [26].

*ForeGraph* [25] is a multi-FPGA TGC accelerator. It divides the graph into Intervals and Blocks, subsequently breaks them down into Sub-Intervals and Sub-Blocks. ForeGraph adopts a destination-first replacement strategy to optimize off-chip I/O. Notably, this reorganization technique is also harnessed in the FPGA-based accelerator FPGA-Cache [5].

*NeuGraph* [72] is a scalable parallel GNN training framework that, employing chunk and block division for efficient processing like GridGraph. By processing edge blocks sequentially, NeuGraph minimizes random access inefficiencies. Its graph-aware dataflow engine intelligently breaks graph data into blocks, enabling parallel execution on GPUs.

*HyGCN* [108] is a classic GCN inference accelerator. It borrows the Interval-and-Shard concept from GraphChi to split graph data in GNNs, assigning vertices within each interval continuous numbers and storing them sequentially. This data reorganization facilitates the sequential caching of feature vectors, resulting in enhanced bandwidth utilization.

### 6.3 GPP Methods for Efficient Communication

Table 7. Works with GPP methods for efficient communication

Effect	Method	Algo.	Works
Reduce Communication Frequency	Partition	TGC	Pregel [74], Giraph [6], Mizan [53], PowerGraph [39], GraphLab [70], Gemini [136], PowerLyra [18]
		GNN	AliGraph [112], Dorylus [94], DistGNN [77], MultiGCN [92]
Reduce Communication Latency	Quantization	GNN	BiFeat [73], QGTC [101]
	Sampling	GNN	AliGraph [112], DistDGL [132], AGL [124], PaGraph [63], P3 [33], 2PGraph [125], DistDGLv2 [133], DGTP [71], Pasca [128], Sugar [106]
		TGC	RabbitOrder [4]

**6.3.1 Reduce Communication Frequency.** Within a parallel graph processing system, reducing the communication frequency between computing components is paramount to mitigating communication overhead. An impactful pre-processing strategy involves enhancing data locality through graph partitioning, consequently decreasing communication demands among computing nodes. The subsequent analysis delves into the mechanics of this approach.

- **Partition.** To decrease communication frequency, effective graph partition must minimize edge cutting, reducing inter-subgraph data dependency. This objective has led to the development of various partition strategies prioritizing enhanced data locality. However, achieving a balance between load distribution and data locality often involves trade-offs. It's important to note that, for power-law graphs, edge-cut partition typically outperforms vertex-cut partition due to its ability to reduce inter-node communication by copying mirror vertices. Below, we introduce some typical works. Other works listed in Table 7 are detailed in Section 6.1.1, and not be reiterated here.

*DistGNN* [77] is a distributed framework, performing GNN training on CPU-clusters. It uses DBH [104] partition, a vertex-cut method to reduce communication frequency.

*MultiGCN* [92], a multi-node GCN accelerator, introduces a round partition method, dividing replicas into processing rounds to store them entirely in on-chip, reducing off-chip data access.

**6.3.2 Reduce Communication Latency.** The high communication latency arises from two factors: extensive communication data volume and irregular communication patterns. To reduce latency, it can be tackled on two fronts. Firstly, data scale can be curtailed through techniques like data quantization and sampling. Concurrently, optimizing data layout via reorder can ameliorate irregular communication, ultimately enhancing the utilization of communication bandwidth.

- **Quantization.** This method produces a more compact representation, reducing transmitted data amount during communication and then mitigating communication latency. This process skillfully balances data size reduction with preserving essential information for accurate computation, bolstering the efficiency of distributed graph processing. Notably, several distributed GNN frameworks have integrated quantization, including BiFeat [73] and QGTC [101].

- **Sampling.** Through sampling mini-batches, the communication data volume between computing components can be diminished, thus reducing synchronization latency. Compared to full-batch training, sampling-based training significantly reduces communication overhead, enabling more efficient parallelism, faster convergence, and the ability to handle larger datasets. Table 7 lists sampling-based parallel frameworks.

- **Reorder.** This method reduces inter-node communication by grouping related vertices and edges. Reordered layouts simplify irregular communication and enhance data access efficiency, thus lowering the resources and time spent on communication and mitigating communication latency in distributed graph processing. An example is the RabbitOrder [4] framework, which employs graph reorder to reduce communication overhead.

## 7 SUMMARY AND COMPARISON

The preceding sections have meticulously explored GPP methods through the lenses of algorithmic principles and hardware optimization, while showcasing illustrative examples. This section provides a comprehensive summary and comparison of seven GPP methods, as outlined in Tables 8 and 9.

- **TGC vs. GNN:** GPP methods for TGC leverage graph characteristics to optimize performance. Among these methods, partition evenly divides large-scale graphs, while reorder and reorganization enhance data access, considering factors like graph irregularity and the power-law distribution. Importantly, partition, reorder, and reorganization are versatile techniques not bound to

Table 8. Summary of GPP methods in algorithmic optimization

Method	TGC	GNN		Information Loss	Accuracy Sacrifice
	Algorithm	Backbone	Phase		
Partition	All	All	Training & Inference	✓	✗
Sampling	-	GCN, GAT	Training	✓	✗
Sparsification	-	GCN, GAT, GIN	Training & Inference	✓	✗
Reconstruction	-	GCN, GAT	Training & Inference	✗	✗
Quantization	-	All	Training	✓	✓
Reorder	All	GCN, GAT	Training & Inference	✗	✗
Reorganization	All	GCN, GAT	Training & Inference	✗	✗

Table 9. Summary of GPP methods in hardware optimization

Method	Optimization Effect		GFP Platform
Parition	Computing → Load Balance	CPU, GPU, FPGA, ASIC, PIM	
	Communication → Reduce Communication Frequency		
Sampling	Computing → Reduce Computation	CPU, GPU, FPGA, ASIC, PIM	
	Storage → Reduce Capacity Requirement		
	Communication → Reduce Communication Latency		
Sparsification	Computing → Reduce Computation	CPU, GPU	
Reconstruction	Computing → Reduce Computation	CPU, GPU, FPGA, ASIC	
	Storage → Reduce Data Movement		
Quantization	Computing → Reduce Computation	CPU, GPU, FPGA	
	Storage → Reduce Capacity Requirement		
	Communication → Reduce Communication Latency	CPU, GPU	
Reorder	Computing → Reduce Computation	PIM	
	Storage → Reduce Data Movement		
	Communication → Reduce Communication Latency	CPU, GPU, FPGA, ASIC	
Reorganization	Storage → Reduce Data Movement	CPU, GPU, FPGA, ASIC, PIM	

specific TGC algorithms. These well-established GPP methods have the potential to optimize the execution of all TGC algorithms.

GPP methods for GNNs take into account the specific characteristics of GNN models. Among these methods, reconstruction aims to eliminate redundancy in aggregation, thereby enhancing data utilization. Other methods are derived from technologies used in TGC, such as partition, reorder, and reorganization, or from NNs, including sampling, sparsification, and quantization. This integration is due to GNNs combining graph structures and NNs. These methods are adapted and optimized to align with the unique requirements of GNNs. Notably, sampling and quantization are GPP methods designed explicitly to accelerate GNN training, while other GPP methods can serve both training and inference phase. Nobably, a significant optimization efforts are directed towards ConvGNN models like GCN, GAT, and GIN. Consequently, technologies such as sampling, sparsification, reconstruction, reorder, and reorganization are closely integrated with convolutional models, specifically tailored to meet their unique requirements. While partition and quantization are general GPP methods, their primary application remains the optimization of GCN and GAT execution.

- **Information Loss vs. Accuracy Sacrifice:** Reconstruction, reorder, and reorganization do not incur information loss because they solely change the storage access order of data. In contrast, other GPP methods can result in information loss. These methods include partition, sampling, and sparsification, which involve the removal of edges and vertices, and quantization, which involves precision degradation. It's important to highlight that only quantization techniques are typically associated with potential accuracy sacrifice, since it significantly compresses the precision of data

used for training. Fortunately, given the inherent robustness of NNs, this accuracy sacrifice can often be managed within an acceptable range through refinement and adjustment.

Conversely, while partition may cut vertices and edges, the resulting information loss is generally minimal, preserving accuracy. Sampling methods selecting subgraphs for mini-batches inherently incur information loss, yet well-designed strategies can retain vital structural and feature-related details while still maintaining accuracy. Sparsification, which selectively removing edges from the original graph, inherently results in some loss of information, but it can serve as a data augmentation strategy to alleviate overfitting in deep GNNs and thereby improving training accuracy.

- **Optimization Effect vs. GFP Platform:** Partition and sampling are versatile GPP methods that accelerate a range of graph processing algorithms and yield varied hardware optimization benefits. Among these, partition divides large graphs into manageable subgraphs for parallel processing, aiming for load balance and reduced communication frequency. Balancing these goals presents challenges, requiring trade-offs based on the application. On the other hand, sampling delivers optimization in computing, storage, and communication, particularly for large-scale GNN training. It reduces per-iteration computation, accelerates convergence through smaller subgraphs, and trims memory and communication overhead. Sampling-based mini-batch training is favored for its efficiency and scalability in large-scale GNN scenarios. Notably, many large-scale GNN training scenarios adopt the partition+sampling approach. Careful consideration of sampling locality and overhead is essential during partitioning, which influences method selection.

Sparsification reduces computational demands, speeding up execution by removing redundant edges. However, in GNN training, many sparsification techniques prioritize accuracy improvements. While reducing per-iteration time through data reduction, the elevated iteration count for convergence might lead to an overall runtime increase. Consequently, existing GNN works are mostly using sparsification for accuracy enhancement and are commonly deployed on general platforms such as CPUs and GPUs.

Graph reconstruction proves efficient in curtailing redundant computations and data movements through the reuse of intermediate results. This approach involves minimizing repetitive operations and optimizing the utilization of computing resources, which holds particular advantages for customized architectures, resulting in substantial resource conservation. In heterogeneous architectures, where pre-processing occurs on CPUs, graph reconstruction helps mitigate data transfers between CPUs and specialized platforms like FPGAs and ASICs, further hastening GNN execution. On general platforms such as CPUs and GPUs, graph processing frameworks store reusable vertices in cache and directly access previously calculated results when recalculation is required, thereby achieving acceleration.

Quantization offers a notable advantage through significant memory reduction achieved via data compression, a feature particularly beneficial for memory-constrained edge devices. Moreover, data compression contributes to the reduction of communication overhead in distributed GNN computation, since transmitting quantized data across nodes requires fewer bits, leading to decreased communication latency and alleviating network congestion. As a result, quantization is primarily employed in CPU or GPU-based large-scale graph processing frameworks. Furthermore, representing data with fewer bits simplifies complex floating-point operations into shifts, conserving computing resources and energy, especially when applied to FPGAs.

Reorder enhances data locality by grouping frequently accessed data together, minimizing the need for retrieval from distant memory locations and unnecessary data movement. This optimization is particularly beneficial when irregular memory access patterns lead to high cache misses and

inefficient data transfer. In parallel graph processing, reorder improves data locality, reduces irregular node communication, and decreases communication latency. Reorder is applicable across various platforms, including CPU, GPU, FPGA, and ASIC. Additionally, for PIM-based architectures, reorder enhances computing efficiency for ReRAM by reducing zero elements in matrices.

Reorganization is a fundamental data flow optimization technique employed to enhance memory access efficiency. It achieves this by transforming irregular memory accesses into contiguous ones, effectively reducing data movement resulting from non-sequential data access. This optimization technique finds widespread use in graph processing implementations across a range of platforms, including CPU, GPU, FPGA, ASIC, and PIM.

## 8 CHALLENGES AND FUTURE DIRECTIONS

In previous sections, we extensively discussed various GPP methods from algorithmic and hardware standpoints, demonstrating their efficacy in accelerating graph processing applications. However, challenges persist due to algorithmic and device-related factors. In this section, we explore challenges within GPP and discuss potential future directions.

### 8.1 Challenges of GPP

Currently, the field of GPP faces four primary challenges:

- **High Overhead:** As data scales continue to increase, the time overhead required for GPP becomes more pronounced. For instance, graph reorder time that takes 13.4 seconds on the Pokec dataset can take as long as 1.5 hours on the Twitter dataset [102]. While certain heuristics like Hub-Sorting and Hub-Clustering have been introduced to reduce reorder time, these methods may fall short compared to more intricate techniques, potentially affecting result accuracy. Therefore, there is an urgent need to develop efficient and optimized GPP approaches tailored for large-scale graphs.

- **Dynamic Graphs:** Dynamic graph processing has gained popularity [75, 126], and GPP techniques designed for static graphs may not directly apply to dynamic graphs that evolve over time. More online GPP techniques that can adapt to dynamic graphs and minimize the GPP overhead are needed. In scenarios where dynamic graph data arrives in a streaming fashion, GPP methods must efficiently process and update the graph representation on-the-fly. This requires designing algorithms that can handle the data stream efficiently and make real-time decisions.

- **Accuracy Loss:** GPP methods such as quantization and sampling are commonly employed to improve execution efficiency, but they can introduce precision loss [66, 69]. Although advancements have been made in mitigating accuracy loss caused by these techniques, achieving a balance between accuracy and optimization remains a challenge. Future research endeavors should strive to discover innovative GPP methods capable of minimizing precision loss while achieving substantial computational gains. This will enable more effective improvement of overall performance.

- **Heterogeneous Graphs:** While existing GNN methods predominantly focus on isomorphic graphs, the recent emergence of heterogeneous graph models has sparked a new research trend [60, 100, 135, 138]. As heterogeneous graphs consist of different types of edges and vertices [48], GPP techniques designed for isomorphic graphs are not directly applicable. To effectively analyze and process the complex graph structures represented by heterogeneous graphs, it is essential to develop tailored GPP techniques specifically designed for such graphs.

### 8.2 Future Prospects of GPP

- **GPP Profiling:** Addressing the aforementioned challenges requires a thorough assessment of the behavioral characteristics and performance bottlenecks of graph processing and pre-processing. This evaluation provides crucial insights for optimizing execution. Previous efforts have evaluated

some aspects, such as GCN execution and distributed GNN training on GPUs [61, 107], as well as bottlenecks in dynamic and heterogeneous graphs [16, 110]. Nonetheless, a comprehensive analysis and evaluation of GPP remain unexplored, presenting a promising avenue for future research.

- **Pipeline-friendly GPP:** With the expansion of data scales and the escalation of model complexity, the impact of GPP overhead becomes more pronounced, particularly in the realm of online methods. While more intricate GPP algorithms might offer better acceleration, it could potentially introduce pipeline stalls within the entire system. Thus, achieving a delicate trade-off between the provided acceleration and GPP overhead is of paramount importance. An effective approach involves the discerning selection of appropriate GPP methods and the meticulous optimization of the temporal overlap between GPP and GFP steps to attain optimal acceleration. For instance, GraphACT [118] integrates a swift graph reconstruction technique for online GPP, seamlessly integrating with GNN inference execution on FPGA. Furthermore, efforts are channeled towards the improvement of GPP algorithmic execution; for example, GNNSampler [68] implements locality-aware optimizations to sampling algorithms, aimed at curtailing sampling overhead.

- **Accuracy-friendly GPP:** Certain GPP methods have the potential to impact the final accuracy of graph processing. Some works make efforts to mitigate this issue and reduce the influence of GPP on accuracy. For instance, works like SGQuant [32] and DegreeQuant [93] have developed degree-based mechanisms for quantization, specifically designed to minimize precision loss. Another approach, known as EXACT [69], addresses precision loss through projection techniques. It is important to consider that some of these methods may require additional memory space. Striking a balance between accuracy loss and optimization effects while carefully considering the trade-offs involved is crucial in achieving both accurate and efficient graph processing.

- **Comprehensive Framework of GPP:** Optimizing GPP often involves combining multiple GPP methods. In distributed GNN training systems, as discussed in [132] and [95], input graphs are partitioned into subgraphs and sampled for training. Others, such as [120] and SnF [115], utilize reconstruction, reorder, and sampling. Exploring the adaptability and potential conflicts between various GPP methods is crucial for developing a comprehensive and configurable GPP framework. A recent example is EndGraph [65], a distributed pre-processing framework that accelerates graph partition and construction, improving pre-processing performance up to  $35.76\times$  compared to state-of-the-art systems.

- **Specific Accelerators for GPP:** GPP is typically performed on CPUs, even when executing formal processing on GPUs or custom architectures. For instance, CuSha [54] performs partition and reorganization on CPUs before executing traditional graph computing on GPUs. Similarly, heterogeneous platforms like GraFBoost [50] and GraphACT [118] utilize CPU-based GPP tasks before executing graph algorithms on FPGA. However, due to the increasing GPP overhead, it is worthwhile to design dedicated graph GPP accelerators to enhance online GPP and facilitate seamless collaboration with fast-executing rear accelerators. For example, I-GCN [35] has developed specialized architectures for graph reconstruction, resulting in significant improvements in overall processing speed. Implementing such accelerators, GPP tasks can be efficiently handled in real-time, effectively improving the system performance.

- **GPP for Heterogeneous Graphs:** With the increasing popularity of heterogeneous graphs, the optimization of their processing has emerged as a significant research area. Despite existing efforts towards optimizing the execution of such graphs, exemplified by accelerators like HiHGNN [105], there remains a demand for specialized GPP techniques catering to these complex structures. While a few GPP methods designed for heterogeneous graphs already exist, such as HetGNN [122] and HGSampling [47], further research is essential to develop more advanced GPP approaches.

## 9 CONCLUSION

Graph pre-processing (GPP) is a crucial step that transforms raw graph data to prepare it for the formal execution of graph processing algorithms. GPP is widely used across various execution systems, including general frameworks on CPUs and GPUs, as well as custom accelerators based on FPGA, ASIC, PIM, and more. However, as graph data scales up, the overhead of GPP becomes significant, necessitating thorough analysis and optimization of GPP.

In this paper, we present a comprehensive survey of GPP methods from both algorithmic and hardware perspectives, aiming to provide valuable insights for optimizing graph processing algorithms and hardware acceleration. We discuss the challenges in graph processing execution and emphasize the critical role of GPP in graph processing. Existing GPP methods can be concluded into seven types: partition, sampling, sparsification, reconstruction, quantization, reorder, and reorganization. Our taxonomy of GPP methods comprises two decision levels, where algorithmic categories include graph representation and data representation optimization, while hardware categories encompass efficient computation, storage, and communication.

Finally, we provide a summary and comparison of GPP methods, discussing challenges and potential future directions. Despite facing challenges such as high overhead and adaptability concerns, certain endeavors have been undertaken to enhance GPP. Many GPP techniques hold an irreplaceable position in heightening the efficiency of various graph applications on diverse platforms. We await increased attention and exploration of GPP.

## ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program under Grant 2022YFB4501404, in part by the National Natural Science Foundation of China under Grant 62202451, the CAS Project for Young Scientists in Basic Research under Grant YSBR-029, and the CAS Project for Youth Innovation Promotion Association.

## REFERENCES

- [1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2021. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–38.
- [2] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming graph partitioning: an experimental study. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1590–1603.
- [3] Hamilton Wilfried Yves Adoni, Tarik Nahhal, Moez Krichen, Brahim Aghezzaf, and Abdeltif Elbyed. 2020. A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems. *Distributed and Parallel Databases* 38 (2020), 495–530.
- [4] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamoto, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 22–31.
- [5] Mikhail Asiatici and Paolo Innen. 2021. Large-scale graph processing on FPGAs with caches for thousands of simultaneous misses. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 609–622.
- [6] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit, Santa Clara* 11, 3 (2011), 5–9.
- [7] Ariful Azad, Mathias Jacquelain, Aydin Buluç, and Esmond G Ng. 2017. The reverse Cuthill-McKee algorithm in distributed-memory. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 22–31.
- [8] Mehdi Bahri, Gaétan Bahl, and Stefanos Zafeiriou. 2021. Binary graph neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 9492–9501.
- [9] Jiyang Bai, Yuxiang Ren, and Jiawei Zhang. 2021. Ripple walk training: A subgraph-based training framework for large and deep graph neural network. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.
- [10] Vignesh Balaji and Brandon Lucia. 2018. When is graph reordering an optimization? studying the effect of light-weight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 203–214.

- [11] MACIEJ BESTA, DIMITRI STANOJEVIC, JOHANNES DE FINE LICHT, TAL BEN-NUN, and TORSTEN HOEFLER. 2019. Graph Processing on FPGAs: Taxonomy, Survey, Challenges. *arXiv preprint arXiv:1903.06697* (2019).
- [12] Florian Bourne, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1456–1465.
- [13] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 185–195.
- [14] Cen Chen, Kenli Li, Yangfan Li, and Xiaofeng Zou. 2022. ReGNN: A redundancy-eliminated graph neural networks accelerator. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 429–443.
- [15] Cen Chen, Kenli Li, Xiaofeng Zou, and Yangfan Li. 2021. DyGNN: Algorithm and Architecture Support of Dynamic Pruning for Graph Neural Networks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 1201–1206.
- [16] Hanqiu Chen, Yahya Alhinai, Yihan Jiang, Eunjee Na, and Cong Hao. 2022. Bottleneck Analysis of Dynamic Graph Neural Network Inference on CPU and GPU. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 130–145.
- [17] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*.
- [18] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.
- [19] Tianlong Chen, Yongduo Sui, Xuxi Chen, Aston Zhang, and Zhangyang Wang. 2021. A unified lottery ticket hypothesis for graph neural networks. In *International Conference on Machine Learning*. PMLR, 1695–1706.
- [20] Xiaobing Chen, Yuke Wang, Xinfeng Xie, Xing Hu, Abanti Basak, Ling Liang, Mingyu Yan, Lei Deng, Yufei Ding, Zidong Du, et al. 2021. Rubik: A hierarchical architecture for efficient graph neural network training. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 4 (2021), 936–949.
- [21] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2018. Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Processing Magazine* 35, 1 (2018), 126–136.
- [22] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. Nxgraph: An efficient graph processing system on a single machine. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 409–420.
- [23] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 257–266.
- [24] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph processing framework on FPGA a case study of breadth-first search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 105–110.
- [25] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 217–226.
- [26] Guohao Dai, Tianhao Huang, Yu Wang, Huazhong Yang, and John Wawrzynek. 2019. GraphSAR: A sparsity-aware processing-in-memory architecture for large-scale graph processing on ReRAMs. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 120–126.
- [27] Hanjun Dai, Zornitsa Kozareva, Bo Dai, Alex Smola, and Le Song. 2018. Learning steady-states of iterative algorithms over graphs. In *International conference on machine learning*. PMLR, 1106–1114.
- [28] Austin Derrow-Pinion, Jennifer She, David Wong, Oliver Lange, Todd Hester, Luis Perez, Marc Nunkesser, Seongjae Lee, Xueying Guo, Brett Wiltshire, et al. 2021. Eta prediction with graph neural networks in google maps. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 3767–3776.
- [29] Iain S Duff. 1984. Computer Solution of Large Sparse Positive Definite Systems (Alan George and Joseph W. Liu. *SIAM Rev.* 26, 2 (1984), 289–291.
- [30] Priyank Faldu, Jeff Diamond, and Boris Grot. 2019. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–13.
- [31] Xiaomin Fang, Jizhou Huang, Fan Wang, Lingke Zeng, Haijin Liang, and Haifeng Wang. 2020. Constgat: Contextual spatial-temporal graph attention network for travel time estimation at baidu maps. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2697–2705.
- [32] Boyuan Feng, Yuke Wang, Xu Li, Shu Yang, Xueqiao Peng, and Yufei Ding. 2020. Sqquant: Squeezing the last bit on graph neural networks with specialized quantization. In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 1044–1052.

- [33] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed deep graph learning at scale. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 551–568.
- [34] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. 2020. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 922–936.
- [35] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. 2021. I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1051–1063.
- [36] Seyed Ali Ghasemi, Belal Jahannia, and Hamed Farbeh. 2022. GraphA:: An efficient ReRAM-based architecture to accelerate large scale graph processing. (2022).
- [37] Vladimir Gligorijević, P Douglas Renfrew, Tomasz Kosciolek, Julia Koehler Leman, Daniel Berenberg, Tommi Vatanen, Chris Chandler, Bryn C Taylor, Ian M Fisk, Hera Vlamakis, et al. 2021. Structure-based protein function prediction using graph convolutional networks. *Nature communications* 12, 1 (2021), 1–14.
- [38] Zhangxiaowen Gong, Houxiang Ji, Yao Yao, Christopher W Fletcher, Christopher J Hughes, and Josep Torrellas. 2022. Graphite: optimizing graph neural networks on CPUs through cooperative software-hardware techniques. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 916–931.
- [39] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. {PowerGraph}: Distributed {Graph-Parallel} Computation on Natural Graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.
- [40] CY Gui, L Zheng, BS He, et al. 2019. A Survey on Graph Processing Accelerators: Challenges and Opportunities. *J. Comput. Sci. & Technol* 34, 2 (2019).
- [41] Yilong Guo, Yuxuan Chen, Xiaofeng Zou, Xulei Yang, and Yuandong Gu. 2022. Algorithms and architecture support of degree-based quantization for graph neural networks. *Journal of Systems Architecture* (2022), 102578.
- [42] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.
- [43] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).
- [44] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 77–85.
- [45] Safiollah Heidari, Yogesh Simmhan, Rodrigo N Calheiros, and Rajkumar Buyya. 2018. Scalable graph processing frameworks: A taxonomy and open challenges. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–53.
- [46] Drahomira Herrmannova and Petr Knoth. 2016. An analysis of the microsoft academic graph. *D-lib Magazine* 22, 9/10 (2016), 1.
- [47] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020*. 2704–2710.
- [48] Rana Hussein, Dingqi Yang, and Philippe Cudré-Mauroux. 2018. Are meta-paths necessary? Revisiting heterogeneous graph embeddings. In *Proceedings of the 27th ACM international conference on information and knowledge management*. 437–446.
- [49] Zhihao Jia, Sina Lin, Rex Ying, Jiaxuan You, Jure Leskovec, and Alex Aiken. 2020. Redundancy-free computation for graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 997–1005.
- [50] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. 2018. GraFBoost: Using accelerated flash storage for external graph analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 411–424.
- [51] Konstantinos I Karantasis, Andrew Lenhardt, Donald Nguyen, Mara J Garzaran, and Keshav Pingali. 2014. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 921–932.
- [52] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.
- [53] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European conference on computer systems*. 169–182.
- [54] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 239–252.

- [55] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi:{Large-Scale} Graph Computation on Just a {PC}. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 31–46.
- [56] Kartik Lakhotia, Shreyas Singapura, Rajgopal Kannan, and Viktor Prasanna. 2017. Recall: Reordered cache aware locality based graph processing. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, 273–282.
- [57] Dominique LaSalle and George Karypis. 2013. Multi-threaded graph partitioning. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 225–236.
- [58] Dongyue Li, Tao Yang, Lun Du, Zhezhi He, and Li Jiang. 2021. Adaptivegcn: Efficient gcn through adaptively sparsifying graphs. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 3206–3210.
- [59] Shun Li, Yuxuan Tao, Enhao Tang, Ting Xie, and Ruiqi Chen. 2022. A survey of field programmable gate array (FPGA)-based graph convolutional neural network accelerators: challenges and opportunities. *PeerJ Computer Science* 8 (2022), e1166.
- [60] Siyu Li, Jin Yang, Gang Liang, Tianrui Li, and Kui Zhao. 2022. SybilFlyover: Heterogeneous graph-based fake account detection model on social networks. *Knowledge-Based Systems* 258 (2022), 110038.
- [61] Haiyang Lin, Mingyu Yan, Xiaocheng Yang, Mo Zou, Wenming Li, Xiaochun Ye, and Dongrui Fan. 2022. Characterizing and understanding distributed GNN training on GPUs. *IEEE Computer Architecture Letters* 21, 1 (2022), 21–24.
- [62] Haiyang Lin, Mingyu Yan, Xiaochun Ye, Dongrui Fan, Shirui Pan, Wenguang Chen, and Yuan Xie. 2022. A Comprehensive Survey on Distributed Training of Graph Neural Networks. *arXiv preprint arXiv:2211.05368* (2022).
- [63] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN Training on Large Graphs via Computation-Aware Caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing (Virtual Event, USA) (SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 401–415.
- [64] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2023. {BGL}:{GPU-Efficient}{GNN} Training by Optimizing Graph Data {I/O} and Preprocessing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 103–118.
- [65] Tianfeng Liu and Dan Li. 2022. Endgraph: An efficient distributed graph preprocessing system. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 111–121.
- [66] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, and Dongrui Fan. 2021. Sampling methods for efficient training of graph convolutional networks: A survey. *IEEE/CAA Journal of Automatica Sinica* 9, 2 (2021), 205–234.
- [67] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, Dongrui Fan, Shirui Pan, and Yuan Xie. 2022. Survey on Graph Neural Network Acceleration: An Algorithmic Perspective. *arXiv e-prints* (2022), arXiv–2202.
- [68] Xin Liu, Mingyu Yan, Shuhan Song, Zhengyang Lv, Wenming Li, Guangyu Sun, Xiaochun Ye, and Dongrui Fan. 2021. Gnnsampler: Bridging the gap between sampling algorithms of gnn and hardware. *arXiv preprint arXiv:2108.11571* (2021).
- [69] Zirui Liu, Kaixiong Zhou, Fan Yang, Li Li, Rui Chen, and Xia Hu. 2021. EXACT: Scalable graph neural networks training via extreme activation compression. In *International Conference on Learning Representations*.
- [70] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB* 5.
- [71] Ziyue Luo, Yixin Bao, and Chuan Wu. 2022. Optimizing task placement and online scheduling for distributed GNN training acceleration. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 890–899.
- [72] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. {NeuGraph}: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 443–458.
- [73] Yuxin Ma, Ping Gong, Jun Yi, Zhewei Yao, Minjie Wang, Cheng Li, Yuxiong He, and Feng Yan. 2022. BiFeat: Supercharge GNN Training via Graph Feature Quantization. *arXiv e-prints* (2022), arXiv–2207.
- [74] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.
- [75] Franco Manessi, Alessandro Rozza, and Mario Manzo. 2020. Dynamic graph convolutional networks. *Pattern Recognition* 97 (2020), 107000.
- [76] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.
- [77] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayanan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. 2021. Distgnn: Scalable distributed training for large-scale graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

- [78] Seung Won Min, Kun Wu, Mert Hidayetoglu, Jinjun Xiong, Xiang Song, and Wen-mei Hwu. 2022. Graph neural network training and data tiering. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3555–3565.
- [79] Sufikureshi Mondal and Nandini Mukherjee. 2019. A bfs-based pruning algorithm for disease-symptom knowledge graph database. In *Information and Communication Technology for Intelligent Systems: Proceedings of ICTIS 2018, Volume 2*. Springer, 417–426.
- [80] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.
- [81] Masahiro Nakao, Koji Ueno, Katsuki Fujisawa, Yuetsu Kodama, and Mitsuhsisa Sato. 2020. Performance evaluation of supercomputer Fugaku using breadth-first search benchmark in Graph500. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 408–409.
- [82] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. HDRF: Stream-Based Partitioning for Power-Law Graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management* (Melbourne, Australia) (CIKM ’15). Association for Computing Machinery, New York, NY, USA, 243–252. <https://doi.org/10.1145/2806416.2806424>
- [83] Xuehai Qian. 2021. Graph processing and machine learning architectures with emerging memory technologies: a survey. *Science China Information Sciences* 64, 6 (2021), 160401.
- [84] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. 2019. DropEdge: Towards Deep Graph Convolutional Networks on Node Classification. In *International Conference on Learning Representations*.
- [85] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.
- [86] Marco Serafini and Hui Guan. 2021. Scalable graph neural network training: The case for sampling. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 68–76.
- [87] Yingxia Shao, Hongzheng Li, Xizhi Gu, Hongbo Yin, Yawen Li, Xupeng Miao, Wentao Zhang, Bin Cui, and Lei Chen. 2022. Distributed Graph Neural Network Training: A Survey. *arXiv preprint arXiv:2211.00216* (2022).
- [88] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR)* 50, 6 (2018), 1–35.
- [89] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 531–543.
- [90] Rakshith S Srinivasa, Cao Xiao, Lucas Glass, Justin Romberg, and Jimeng Sun. 2020. Fast Graph Attention Networks Using Effective Resistance Based Graph Sparsification. *arXiv e-prints* (2020), arXiv–2006.
- [91] Isabelle Stanton and Gabriel Kliot. 2012. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Beijing, China) (KDD ’12). Association for Computing Machinery, New York, NY, USA, 1222–1230. <https://doi.org/10.1145/2339530.2339722>
- [92] Gongjian Sun, Mingyu Yan, Duo Wang, Han Li, Wenming Li, Xiaochun Ye, Dongrui Fan, and Yuan Xie. 2022. Multi-node Acceleration for Large-scale GCNs. *IEEE Trans. Comput.* 01 (2022), 1–12.
- [93] Shyam A Tailor, Javier Fernandez-Marques, and Nicholas D Lane. 2020. Degree-quant: Quantization-aware training for graph neural networks. *arXiv preprint arXiv:2008.05000* (2020).
- [94] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate {GNN} Training with Distributed {CPU} Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation*.
- [95] Chao Tian, Lingxiao Ma, Zhi Yang, and Yafei Dai. 2020. Pcgcn: Partition-centric processing for accelerating graph convolutional network. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 936–945.
- [96] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*. 333–342.
- [97] Jana Vatter, Ruben Mayer, and Hans-Arno Jacobsen. 2023. The Evolution of Distributed Systems for Graph Neural Networks and their Origin in Graph Processing and Deep Learning: A Survey. *Comput. Surveys* (2023).
- [98] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. 2017. Graph attention networks. *stat* 1050, 20 (2017), 10–48550.
- [99] Junfu Wang, Yunhong Wang, Zhen Yang, Liang Yang, and Yuanfang Guo. 2021. Bi-GCN: Binary Graph Convolutional Network. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, 1561–1570.

- [100] Xiao Wang, Deyu Bo, Chuan Shi, Shaohua Fan, Yanfang Ye, and S Yu Philip. 2022. A survey on heterogeneous graph embedding: methods, techniques, applications and sources. *IEEE Transactions on Big Data* (2022).
- [101] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: accelerating quantized graph neural networks via GPU tensor core. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 107–119.
- [102] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*. 1813–1828.
- [103] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.
- [104] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed Power-Law Graph Computing: Theoretical and Empirical Analysis. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) (NIPS’14). MIT Press, Cambridge, MA, USA, 1673–1681.
- [105] Runzhen Xue, Dengke Han, Mingyu Yan, Mo Zou, Xiaocheng Yang, Duo Wang, Wenming Li, Zhimin Tang, John Kim, Xiaochun Ye, et al. 2023. HiHGNN: Accelerating HGNNs through Parallelism and Data Reusability Exploitation. *arXiv preprint arXiv:2307.12765* (2023).
- [106] Zihui Xue, Yuedong Yang, and Radu Marculescu. 2023. SUGAR: Efficient Subgraph-level Training via Resource-aware Graph Partitioning. *IEEE Trans. Comput.* (2023).
- [107] Mingyu Yan, Zhaodong Chen, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. Characterizing and understanding GCNs on GPU. *IEEE Computer Architecture Letters* 19, 1 (2020), 22–25.
- [108] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 15–29.
- [109] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, et al. 2019. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 615–628.
- [110] Mingyu Yan, Mo Zou, Xiaocheng Yang, Wenming Li, Xiaochun Ye, Dongrui Fan, and Yuan Xie. 2022. Characterizing and Understanding HGNNs on GPUs. *IEEE Computer Architecture Letters* 21, 2 (2022), 69–72.
- [111] Weian Yan, Weiqin Tong, and Xiaoli Zhi. 2020. FPGAN: An FPGA accelerator for graph attention networks with software and hardware co-optimization. *IEEE Access* 8 (2020), 171608–171620.
- [112] Hongxia Yang. 2019. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 3165–3166.
- [113] Rui Ye, Xin Li, Yujie Fang, Hongyu Zang, and Mingzhong Wang. 2019. A Vectorized Relational Graph Convolutional Network for Multi-Relational Network Alignment.. In *IJCAI*. 4135–4141.
- [114] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 974–983.
- [115] Mingi Yoo, Jaeyong Song, Hyeyoon Lee, Joungwoo Lee, Namhyung Kim, Youngsok Kim, and Jinho Lee. 2022. Slice-and-Forge: Making Better Use of Caches for Graph Convolutional Network Accelerators. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 40–53.
- [116] Zhizhi Yu, Di Jin, Ziyang Liu, Dongxiao He, Xiao Wang, Hanghang Tong, and Jiawei Han. 2021. AS-GCN: Adaptive semantic architecture of graph convolutional networks for text-rich networks. In *2021 IEEE International Conference on Data Mining (ICDM)*. IEEE, 837–846.
- [117] Abdulganiyu Abdu Yusuf, Feng Chong, and Mao Xianling. 2022. An analysis of graph convolutional networks and recent datasets for visual question answering. *Artificial Intelligence Review* 55, 8 (2022), 6277–6300.
- [118] Hanqing Zeng and Viktor Prasanna. 2020. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 255–265.
- [119] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931* (2019).
- [120] Bingyi Zhang, Hanqing Zeng, and Viktor Prasanna. 2020. Hardware acceleration of large scale gcn inference. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 61–68.
- [121] Chengming Zhang, Tong Geng, Anqi Guo, Jiamian Tian, Martin Herbordt, Ang Li, and Dingwen Tao. 2022. H-GCN: A Graph Convolutional Network Accelerator on Versal ACAP Architecture. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL 2022)*.
- [122] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V Chawla. 2019. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 793–803.

- [123] Chenzi Zhang, Fan Wei, Qin Liu, Zhihao Gavin Tang, and Zhenguo Li. 2017. Graph Edge Partitioning via Neighborhood Heuristic. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) (KDD '17). Association for Computing Machinery, New York, NY, USA, 605–614. <https://doi.org/10.1145/3097983.3098033>
- [124] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: a scalable system for industrial-purpose graph machine learning. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3125–3137.
- [125] Lizhi Zhang, Zhiquan Lai, Shengwei Li, Yu Tang, Feng Liu, and Dongsheng Li. 2021. 2pgraph: Accelerating gnn training over large graphs on gpu clusters. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 103–113.
- [126] Mengqi Zhang, Shu Wu, Xueli Yu, Qiang Liu, and Liang Wang. 2022. Dynamic graph neural networks for sequential recommendation. *IEEE Transactions on Knowledge and Data Engineering* 35, 5 (2022), 4741–4753.
- [127] Shichang Zhang, Atefeh Sohrabizadeh, Cheng Wan, Zijie Huang, Ziniu Hu, Yewen Wang, Jason Cong, Yizhou Sun, et al. 2023. A Survey on Graph Neural Network Acceleration: Algorithms, Systems, and Customized Hardware. *arXiv preprint arXiv:2306.14052* (2023).
- [128] Wentao Zhang, Yu Shen, Zheyu Lin, Yang Li, Xiaosen Li, Wen Ouyang, Yangyu Tao, Zhi Yang, and Bin Cui. 2022. Pasca: A graph neural architecture search system under the scalable paradigm. In *Proceedings of the ACM Web Conference 2022*. 1817–1828.
- [129] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 293–302.
- [130] Huan Zhao, Xiaogang Xu, Yangqiu Song, Dik Lun Lee, Zhao Chen, and Han Gao. 2019. Ranking users in social networks with motif-based pagerank. *IEEE Transactions on Knowledge and Data Engineering* 33, 5 (2019), 2179–2192.
- [131] Cheng Zheng, Bo Zong, Wei Cheng, Dongjin Song, Jingchao Ni, Wenchao Yu, Haifeng Chen, and Wei Wang. 2020. Robust graph representation learning via neural sparsification. In *International Conference on Machine Learning*. PMLR, 11458–11468.
- [132] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 36–44.
- [133] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4582–4591.
- [134] Long Zheng, Jieshan Zhao, Yu Huang, Qinggang Wang, Zhen Zeng, Jingling Xue, Xiaofei Liao, and Hai Jin. 2020. Spara: An energy-efficient ReRAM-based accelerator for sparse graph analytics applications. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 696–707.
- [135] Xinyi Zhou, Junjie Ye, Chak-Wa Pui, Kun Shao, Guangliang Zhang, Bin Wang, Jianye Hao, Guangyong Chen, and Pheng Ann Heng. 2022. Heterogeneous Graph Neural Network-based Imitation Learning for Gate Sizing Acceleration. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.
- [136] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system.. In *OSDI*, Vol. 16. 301–316.
- [137] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning.. In *USENIX Annual Technical Conference*. 375–386.
- [138] Zhihua Zhu, Xinxin Fan, Xiaokai Chu, and Jingping Bi. 2020. HGCR: A heterogeneous graph convolutional network-based deep learning model toward collective classification. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 1161–1171.
- [139] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-dependent importance sampling for training deep and large graph convolutional networks. *Advances in neural information processing systems* 32 (2019).