AAAS
Intelligent Computing
Volume 2022, Article ID 9806758, 32 pages
https://doi.org/10.34133/2022/9806758

Intelligent Computing
A SCIENCE PARTNER JOURNAL

Review Article

Software Systems Implementation and Domain-Specific Architectures towards Graph Analytics

Hai Jin,[1] Hao Qi,[1] Jin Zhao,[1,2] Xinyu Jiang,[1] Yu Huang,[1,2] Chuangyi Gui,[1] Qinggang Wang,[1] Xinyang Shen,[1] Yi Zhang,[1] Ao Hu,[1] Dan Chen,[1] Chaoqiang Liu,[1] Haifeng Liu,[1] Haiheng He,[1] Xiangyu Ye,[1] Runze Wang,[1] Jingrui Yuan,[1] Pengcheng Yao,[1,2] Yu Zhang,[1,2] Long Zheng,[1,2] and Xiaofei Liao[1]

[1]*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China*
[2]*Zhejiang-HUST Joint Research Center for Graph Processing, Zhejiang Lab, Zhejiang, China*

Correspondence should be addressed to Yu Zhang; zhyu@hust.edu.cn

Received 18 August 2022; Accepted 10 October 2022; Published 29 October 2022

Graph analytics, which mainly includes graph processing, graph mining, and graph learning, has become increasingly important in several domains, including social network analysis, bioinformatics, and machine learning. However, graph analytics applications suffer from poor locality, limited bandwidth, and low parallelism owing to the irregular sparse structure, explosive growth, and dependencies of graph data. To address those challenges, several programming models, execution modes, and messaging strategies are proposed to improve the utilization of traditional hardware and performance. In recent years, novel computing and memory devices have emerged, e.g., HMCs, HBM, and ReRAM, providing massive bandwidth and parallelism resources, making it possible to address bottlenecks in graph applications. To facilitate understanding of the graph analytics domain, our study summarizes and categorizes current software systems implementation and domain-specific architectures. Finally, we discuss the future challenges of graph analytics.

1. Introduction

The amount of graph data that represents relationships is rapidly expanding as a result of the widespread popularization of the Internet, the emergence of the Internet+, the digital transformation of society, and the fast growth of the economy [1]. For example, Facebook's social network contains more than 1.71 billion graph vertices and 100 billion graph edges, according to the most recent statistics it provided in December 2016. The need for connected data analysis is rising at the same time, leading to an increase in the volume of graph-structured data in many significant application sectors, including financial analysis, power system operation, social life, and national security monitoring. To effectively analyze and derive relevant information from this graph data, graph computing technology is quickly being developed. Applications for graph computing are increasingly moving away from traditional binary scenarios and into a variety of kinds, structures, and attributes [2, 3]. Complex graph applications, such as graph mining and graph learning, are continuously developing in addition to traditional graph applications (such as graph processing). In this paper, each of these names (graph processing, graph mining, and graph learning) refers only to a specific class of applications as follows:

Graph Processing. Conventional graph algorithms (such as the PageRank for ranking, the adsorption for video recommendation, the single-source shortest path (SSSP) for road selection, and the connected component for clustering) are designed to process graphs iteratively until convergence. Many operations of such graph algorithms are based on traversal operations and generally focus on performing linear algebra-like computational operations on the graph. Compared with traditional computing models, iterative graph algorithms have rich, efficient, and agile analysis capabilities for relational data, and are widely used in real life. For

example, Google needs to regularly rank the influence of hundreds of millions of web pages on the web, and Facebook needs to iteratively analyze its social network graph to control the structural state of the social network and improve the accuracy of advertising delivery

Graph Mining. Graph mining (such as clique finding (CF), motif counting (MC), and frequent subgraph mining (FSM) aims to discover specific structures or patterns in graphs. In addition to the properties of traditional data mining techniques, graph mining technology is an ideal tool for dealing with complex data structures because of its complex data object relationships and rich data presentation. Knowledge and information acquisition through graph mining has been widely used in various fields, such as social sciences [4, 5], bioinformatics [6, 7], and cheminformatics [8, 9]. Specifically, graph mining can be used to discover structure-content relationships in social media data, to mine community-dense subgraphs, to extract network motifs or significant subgraphs in protein-protein or gene interaction networks, to discover 3D motifs in protein structures or chemical compounds, etc.

Graph Learning. As a typical representative of non-Euclidean spatial data, graphs can characterize the relationships between everything. However, due to the irregularity of graph data, existing deep learning models [10] (which deal with Euclidean Space [11] data and are based on the nature of regularized data) cannot be directly applied to graph structured data. For this reason, graph learning (such as graph neural network (GNN) [12] and graph embedding [13]) was developed. Graph neural networks establish a deep learning framework for non-Euclidean spatial data, and compared to traditional network representation learning, it is able to perform deeper information aggregation operations on graph structures than traditional network representation learning models. Currently, graph neural networks are capable of solving many deep learning tasks, link prediction [14], graph clustering [15], and recommendation systems [16]

The fact that these three classes of graph applications are so widely used motivates us to investigate them. Due to the characteristics of sparsity, power-law distribution, and small-world structure of the graph, graph computing brings a series of challenges to modern computer systems based on control flow architecture, such as low execution efficiency of parallel flow, low locality of memory access, and poor scalability of lock synchronization. Therefore, graph computing has recently been a popular topic for study in both academia and industry.

In order to solve many problems of large-scale graph computing, in recent years, researchers have carried out extensive basic research and key technology research on software systems implementation, which mainly focuses on improvements by software technologies on existing general-purpose hardware platforms, such as single-machine platform and distributed platform. However, there is a significant gap between the general-purpose hardware and the unique characteristics of graph analytics [17, 18]. Domain-specific architectures, which primarily pay attention to hardware acceleration through architecture innovations, are necessary as a potential solution that may fill the gap. We classify the domain-specific architectures' research into three major categories, FPGA, ASIC, and PIM, because different hardware platforms have different considerations for performance acceleration.

This paper will summarize the research status of graph computing key technologies of the software systems implementation and domain-specific architectures, and then summarize, compare, and analyze the latest research progress from three aspects: basic theory, system software, and system architecture. The remainder of this paper is structured as follows: Section 2 explains the background of graph terminology and graph accelerator architecture types. Section 3 describes software systems implementation for graph analytics. Section 4 presents domain-specific architectures for graph analytics. Finally, Section 5 prospects the future technical challenges and research directions, and Section 6 provides a conclusion.

2. Background

2.1. Graph Terminology. A graph is a kind of data structure made up of vertices and the edges that connect vertices. The formula for a graph is $G = (V; E)$, where V stands for the vertex set and E for the edge set. A directed edge from vertex v_i to vertex v_j is represented as $e = (v_i; v_j)$. Each vertex and each edge has its attribute value at the same time. Different domain attribute values can represent different meanings. For instance, in a social network, the attribute value of the vertex is the popularity of the individual, and the attribute value of the edge is expressed as the degree of closeness between two people who are related. The graph data structure expresses the correlation between data well, and correlation computing is the foundation of big data computing. By obtaining the correlation of data, useful information can be extracted from the massive data with a lot of noise. Graph analytics technology solves the problems of low efficiency and high cost of association queries in traditional computing modes and fully characterizes the relationship in the problem domain, and has rich, efficient, and agile data analysis capabilities.

2.2. Domain-Specific Architecture Types for Graph Analytics. FPGA-Based Architecture. Field-programmable gate arrays (FPGAs) are integrated circuits that consist of various types of programmable resources, which enables developers to rapidly prototype application-specific accelerators using dedicated hardware description languages and reconfigure these accelerators as often as needed. These programmable resources include but not limited to lookup tables (LUTs), registers, block RAMs (BRAMs), and DSP slices. However, FPGAs offer reconfigurability at the expense of lowered clock frequencies, which is about $10\times$ lower than that of CPUs. Nevertheless, FPGAs have become attractive devices for accelerating graph applications due to the following advantages.

First, graph-application-specific operations can be elaborately constructed as a pipeline to yield one result per cycle, expressing impressive efficiency. Meanwhile, the pipeline

duplication can be easily implemented on the FPGA, enabling massively parallel graph processing. Furthermore, graph data can be streamed into the pipeline and trigger the computation instead of expensive instruction control operations (e.g., instruction decoding) in modern CPUs and GPUs, reducing power consumption significantly. The random access feature of on-chip BRAMs is another representative advantage of FPGAs, which enables random graph data access with high throughput on FPGAs. What is more, developers can explicitly configure the on-chip BRAMs with the domain-specialized replacement policy, which is fundamentally different from the domain-agnostic cache replacement strategy on CPUs, thereby exploiting locality and reducing off-chip communications effectively. Nowadays, FPGAs have been widely deployed in the cloud or data centers such as Microsoft Project Catapult [19] and Amazon F1 cloud [20]. The accessibility and low cost of FPGAs further make them attractive.

The aforementioned advantages of FPGAs have attracted a good deal of research in developing FPGA-based specialized architecture for accelerating graph application. To mitigate performance degradation caused by irregular access characteristics of graph applications, numerous previous studies focus on designing an efficient memory subsystem [17, 21–28]. Some [21–23] adopt dedicated on-chip data replacement and sophisticated graph partitioning schemes to enhance data reuse and improve locality. Some [17, 24, 25] further alleviate the performance impact of data conflicts occurring in the on-chip BRAM. There are also works aiming to efficiently utilize the bandwidth of on-chip and off-chip memories [26–28]. Recently, emerging 3D-stacked memories, e.g., hybrid memory cube (HMC) [29] and high-bandwidth memory (HBM) [30], take the place of commodity memories to boost the power of FPGAs. Some research efforts [31–34] try to exploit the high bandwidth and parallelism of these new devices for accelerating graph application. In addition, in order to support large-scale graph applications, a number of studies construct the multi-FPGA architecture [21, 22]. Alternatives [35–39] employ the CPU-FPGA heterogeneous platform to handle large graphs using two distinct design methodologies. The first category [35, 36, 40] uses the CPU for data preprocessing and task scheduling while the FPGA is responsible for the real computation. Another category [37–39] features CPU-FPGA coprocessing to release the performance potential of heterogeneous platforms.

ASIC-based architecture. Application specific integrated circuit (ASIC) is an integrated circuit customized for specific requirements. It adopts a certain process to interconnect wirings and components (e.g., transistors, resistors, capacitors, and inductors), manufacture them on one or several small semiconductor wafers or dielectric substrates, and then encapsulate them in a tube to become microstructures with specific circuit functions.

Existing ASIC-based accelerators for graph applications typically focus on elaborately constructing application-specific computation units [41–44] and memory hierarchy [45–47] for higher performance and energy efficiency. For example, in the aspect of dedicated computation units, Graphicionado [48] builds graph-processing-friendly pipelines to enable efficient pipelining computations. HyGCN [49] establishes the hybrid execution engines to alleviate irregularity of the aggregation phase and exploits regularity in the combination phase for graph convolutional neural networks. As for the memory hierarchy, GRAMER [50] architects a locality-aware on-chip memory hierarchy, which can handle the substantial random accesses appearing in graph mining applications to minimize the off-chip communications. Ozdal et al. [51] designed dedicated caches for different types of graph data according to the access characteristics. In addition, recent studies aim at exposing flexibility [51–53] and releasing productivity [51, 54, 55] for ASIC-based graph accelerators.

PIM-Based Architecture. Processing-In-Memory (PIM) is a promising technology that addresses the "memory wall" challenge. The key idea is to move the compute units inside the memory, so that the latency and energy consumption of data movement are drastically reduced compared to the conventional von Neumann architecture with a separate computation-memory hierarchy. Existing approaches to enable and implement PIM can fall into two categories: processing using memory (PUM) [56–58] and processing near memory (PNM) [59–63].

PUM enables memory chips to have the computing ability by exploiting intrinsic operational principles of the memory circuitry [64–66]. Taking resistive random access memory (ReRAM) as an illustration, a ReRAM cell with low read latency and high energy efficiency has an oxide layer sandwiched between two electrodes [67] to store information by changing the resistance across the oxide layer (Figure 1(b)). Many ReRAM cells are organized as an area-efficient crossbar structure to enable high parallelism and memory capacity (Figure 1(a)). ReRAM can perform a matrix-vector multiplication (MVM) operation [68] at one cycle. Specifically, the information stored in the ReRAM cells is programmed to be conductance $G_{i,j}$, where conductance is the inverses of resistance and i (j) indicates the wordline (bitline). Digital-to-Analog-Converters (DACs) can convert the input data to analog voltages V_i, which are applied to the corresponding wordline. Then the current $Vi \cdot G_{i,j}$ passes through the cell (i, j) into the bitline. Finally, output currents on the same bitline can be accumulated via $I_j = \sum_i V_i \cdot G_{i,j}$, and Analog-to-Digital-Converters (ADCs) further convert the results to digital values. Previous studies mainly focus on addressing limited parallelism [69–71] and reducing superfluous ineffectual computations [72–75] for graph applications.

PNM integrates computational logics (e.g., simple in-order cores) inside or nearby the memory. As a representative, Hybrid Memory Cube (HMC) has a logic layer underneath 3D-stacked memories through-silicon vias (TSVs) [76] as shown in Figure 2. Multiple layers of memory and the bottom logic layer are connected together via TSVs, which offer significantly more internal memory bandwidth than the traditional memory channel. Each memory layer contains multiple banks. A vertically connected stack of several banks from different memory layers is called as a vault

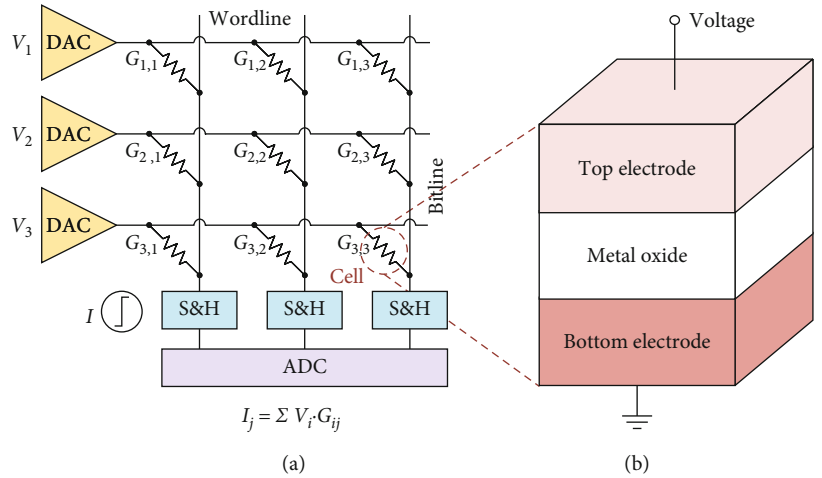

Figure 1: Illustration of the ReRAM architecture.

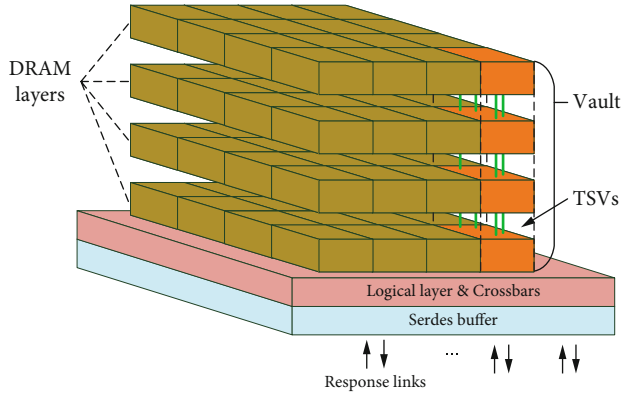

Figure 2: Illustration of the HMC architecture.

[77]. HMC can benefit from multiple DRAM channels for each vault, exhibiting significantly high memory-level parallelism. Plenty of PNM-based accelerators for graph applications are proposed to exploit the massive parallelism [78–80] and reduce communications [81–83].

3. Software Systems Implementation for Graph Analytics

3.1. Software Graph Processing Systems. Many software systems for graph processing are explored on modern general-purpose hardware platforms, and can be classified into two main categories: single-machine graph processing systems and distributed graph processing systems. According to whether the graph data can be stored in memory during processing, these systems can be divided into in-memory graph processing systems and out-of-core graph processing systems. Table 1 summarizes the typical software systems for graph processing. There are four programming models, namely vertex-centric (V), edge-centric (E), path-centric (P), data-centric (D), and two execution models, namely, synchronous (Sync), and asynchronous (Async).

3.1.1. Single-Machine Graph Processing Systems. Single-machine graph processing systems can fully exploit the ability of a single machine to handle graph computation tasks and avoid the expensive network communication overhead in distributed systems. However, such systems are limited by fixed hardware resources and are unable to achieve good scalability, and processing time is typically proportional to the size of the graph data. There are two types of single-machine graph processing systems: in-memory graph processing systems for high-end multicore, large-memory servers, and out-of-core graph processing systems for commercial PCs. The former puts graph data completely into memory during processing, while the latter usually uses disk to store graph data and adopts a certain partitioning strategy to process it in chunks.

Single-machine in-memory graph processing systems often have multiple cores and support very large memory of more than 1 TB, allowing them to handle graph data with hundreds of billions of edges. Compared to single out-of-core graph processing systems, single in-memory graph processing systems keep graph data in memory and can significantly minimize disk I/O overhead. However, single shared memory systems can only scale by increasing the number of CPUs or expanding the memory size.

Ligra [84] is a lightweight shared memory-based single-machine graph processing system, which provides programming abstraction based on edgeMap function, vertexMap function, and vertexSubset type, simplifying the writing of graph processing algorithms. The key idea of Ligra is to accelerate the convergence of the graph algorithms by dynamically switching between the pull and push computation modes during execution based on the size and out-degree of the active vertex subset, but Ligra lacks support for scheduling policies.

The key idea of Galois [85] single-machine graph processing system is to fully exploit the benefits of autonomous scheduling in a data-driven computing mode. Galois designs a machine topology-aware task scheduler and a priority task scheduler with corresponding extension libraries and runtime systems. Galois' flexible and comprehensive programming

Table 1: Overview of typical software systems for graph processing. (IM, PM, EM, DG represent in-memory, programming model, execution model, dynamic graph, respectively, and single machine default to CPU).

Year	System	Architecture	IM	PM	EM	DG	Main features
2013	Ligra [84]	Single machine	Yes	V	Sync	No	Hybrid computing
2013	Galois [85]	Single machine	Yes	V	Async	No	Priority scheduler
2015	Polymer [86]	Single machine	Yes	V	Sync	No	NUMA-aware processing
2017	HotGraph [87]	Single machine	Yes	V	Async	No	Hot graph
2018	CGraph [88, 89]	Single machine	Yes	V	Sync	No	Correlations-aware
2019	GraphBolt [90]	Single machine	Yes	V	Sync	Yes	Incremental computation
2021	DZiG [91]	Single machine	Yes	V	Sync	Yes	DelZero-aware processing
2021	Tripoline [92]	Single machine	Yes	V	Async	Yes	Triangle inequality
2016	Gunrock [93]	Single GPU	Yes	D	Sync	No	Data-centric PM
2019	DiGraph [94]	Multiple GPUs	Yes	V	Async	No	Dependency-aware processing
2020	Scaph [95]	Single GPU	Yes	V	Sync	No	Value-driven scheduling
2012	GraphChi [96]	Single machine	No	V	Async	No	Parallel sliding windows
2013	X-Stream [97]	Single machine	No	E	Sync	No	Edge-centric PM
2015	GridGraph [98]	Single machine	No	E	Async	No	Two-level graph partition
2016	PathGraph [99]	Single machine	No	P	Async	No	Path-centric PM
2017	Mosaic [100]	Single machine	No	V/E	Sync	No	Hilbert-ordered tiles
2019	GraphM [101]	Single machine	No	E	Sync	No	Regularizing traversal path
2022	EGraph [102]	Single GPU	No	E	Sync	Yes	LPS execution model
2010	Pregel [103]	Distributed CPUs	Yes	V	Sync	No	Vertex-centric PM
2012	GraphLab [104]	Distributed CPUs	Yes	V	Both	No	Asynchronous execution
2012	PowerGraph [105]	Distributed CPUs	Yes	V	Both	No	GAS model
2015	PowerSwitch [106]	Distributed CPUs	Yes	V	Both	No	Hybrid computing model
2014	Maiter [107]	Distributed CPUs	Yes	V	Async	No	DAIC theory
2017	KickStarter [108]	Distributed CPUs	Yes	V	Async	Yes	Trimmed approximations
2021	Ingress [109]	Distributed CPUs	Yes	V	Sync	Yes	Flexible memorization
2015	Chaos [110]	Distributed CPUs	No	E	Sync	No	Scale-out graph processing

interfaces allow users to build complicated algorithms as easily as feasible.

Polymer [86] is a multicore processing-oriented nonuniform memory access (NUMA) aware graph processing system. The system performs differential allocation of topology data, application data, and system variable runtime state according to the access pattern to reduce remote memory access while converting random remote access to sequential remote access with lightweight vertex replication across NUMA nodes. Furthermore, Polymer further builds a hierarchical barrier to improve parallelism and locality and uses edge-based balanced partitioning strategies and adaptive data structures to improve load balancing.

HotGraph [87] presents an asynchronous graph processing technique based on core graphs to fully utilize the cascade effect and accelerate the convergence of asynchronous graph algorithms. It contains core graph vertices in the graph, or hot vertices, and the paths between them in a data structure called the core graph. Then, HotGraph gives this core graph a high processing priority, so that the state push in the core graph happens faster. On this basis, in order to speed up the local convergence speed of each graph block, HotGraph adopts an alternate data processing strategy to process each graph block.

CGraph [88, 89] proposes an association-aware execution model and a scheduling algorithm based on core subgraphs, which enables concurrent iterative graph processing jobs to effectively share graph structure data and access in cache/memory and effectively reduce the memory access/computation ratio of concurrent iterative graph processing jobs so as to efficiently execute concurrent iterative graph processing jobs and enable the system to obtain higher throughput.

For dynamic graph processing, incremental computation techniques are usually used. GraphBolt [90] incrementally corrects the difference between the original graph result and the real graph for each round by recording the vertex state of one round during the iterative process and combining it with the dependencies on the graph. The vertex state correction, through dependencies, enables GraphBolt to reduce a large number of redundant computations. However, its need to record each round of vertex state for incremental computation also brings a huge storage overhead.

DZiG [91] notes that many incremental graph processing systems are designed with change-driven models in order to reduce redundant computations, which can lead to sparsity in iterative computations. DZiG adapts to graph changes through a sparsity-aware incremental processing

method, while being able to adaptively switch incremental strategies based on the sparsity of the computation. Compared to GraphBolt, DZiG further improves the performance and increases the scale that can handle graph changes simultaneously.

Tripoline [92] points out that existing incremental graph processing systems often rely on a priori knowledge in terms of queries. For instance, KickStarter's [108] incremental queries rely on the source points not changing each time when they are queried, otherwise the dependency trees they maintain would be useless. Tripoline is able to reuse the latter to speed up the former by establishing strict constraints between the evaluation of one graph query and the result of another graph query, which will not depend on any prior knowledge. GPUs have many processing units and abundant bandwidth resources, which can provide a higher parallel computing capability than CPUs, and can efficiently support large-scale graph vertex traversal and update. High concurrency is a characteristic of graph processing. Both the vertex-centric and edge-centric graph computing programming models hide a large amount of data parallel semantics, enabling GPU parallel acceleration. Graph processing is also a data-intensive application, and the bandwidth of hundreds of GB/S provided by GPUs has obvious advantages over CPUs. More graph data can be transmitted per unit time, and the parallel advantages of GPUs can be fully utilized for acceleration. However, GPU-based graph processing acceleration technologies face challenges such as unbalanced workload, low bandwidth utilization, and insufficient memory capacity.

Gunrock [93] designs a general accelerated library for GPU-based graph processing, and proposes a data-centric programming abstraction that combines high-performance GPU computing primitives with optimization strategies for high-level programming models, enabling fast implementation of high-performance graph primitives on GPUs. Gunrock employs a hybrid scheduling strategy to achieve load balancing of computational tasks across different granularities. At the same time, Gunrock implements a hybrid data structure of CSR and edge list to improve the efficiency of aggregated accesses and reduce the extra overhead caused by random accesses.

DiGraph [94] proposes a path-based multi-GPU acceleration method, which represents a directed graph as a set of disjoint directed paths and treats the paths as the basic parallel processing units, enabling efficient propagation of vertex states along the paths under GPU acceleration, thus speeding up convergence. DiGraph also includes a path-dependent perception scheduling strategy, which processes the paths according to the topological order of the path-dependent graph, effectively reducing the redundant processing of graph data and accelerating the convergence of the graph algorithms.

For faster iterative graph processing on GPUs, AsynGraph [111] proposes a graph structure-aware asynchronous processing method and a forward-backward path processing mechanism to maximize data parallelism for graph processing on GPUs. The former can efficiently perform parallel state propagation for most vertices on the GPU and obtain higher GPU utilization by efficiently processing paths between important graph vertices; the latter can process graph vertices on each path asynchronously, which in turn further increases the speed of state propagation along the path while ensuring lower data access costs.

Scaph [95] is a value-driven scalable GPU-accelerated graph processing system that can effectively improve the utilization of GPU bandwidth by differentially scheduling the processing based on the values that partition the subgraphs. Due to Scaph's excellent scalability, its performance advantage can be progressively increased as computational resources become more readily available.

Subway [112] proposes a fast subgraph generation algorithm, which generates subgraphs of the graph data to be processed quickly by GPU acceleration before each iteration, and then loads the subgraphs into the GPU for processing, thus effectively reducing the data transfer overhead between the CPU and GPU. Additionally, Subway reduces the number of data transfers by delaying the synchronization between the subgraph data in the GPU memory and the graph data in the CPU memory, which further improves the graph processing performance.

With the rapid expansion of graph data size, many single-machine graph processing systems use external memory, such as disks, to store very large scale graph data. The I/O bandwidth limitation of external memory such as disks has become the performance bottleneck of single-machine out-of-core graph processing systems, and it is a challenge to reduce random accesses during graph computing.

GraphChi [96] is the first disk-based graph processing system that proposes the Parallel Sliding Windows (PSW) technique to optimize the access to disk during graph computation. GraphChi preprocesses edge data into shards by a specific graph partitioning strategy, and then the PSW asynchronous computation model is used to process the shards, thus effectively reducing random disk accesses and improving system performance.

X-Stream [97] proposes an edge-centric computation idea to process edges in external memory or in-memory in a streaming manner to improve the continuity of access to memory, thus making full use of the bandwidth of storage devices to improve performance. X-Stream designs a streaming partition mechanism to divide graph data, and then uses an edge-centric scatter-gather computing model to stream the graph partitions, maximizing throughput through streaming access to edges.

GridGraph [98] proposes a two-level graph partitioning strategy, which divides the graph data finely in the preprocessing stage, and then further divides the edge data dynamically at runtime to improve memory access efficiency. At the same time, GridGraph uses dual sliding window technology to stream edge data to reduce the I/O required for computation. In addition, GridGraph also provides a flexible and selective scheduling strategy, which can further reduce graph data I/O.

PathGraph [99] proposes a path-centric graph computing model, which can effectively improve the locality of memory and disk when executing iterative graph algorithms on large-scale graphs. Moreover, the path-centric compressed

storage structure further improves the continuity of data access, thereby accelerating the execution of graph computing tasks.

Mosaic [100] is a heterogeneous graph processing system that scales horizontally and vertically through a hybrid execution model. The main processor is responsible for vertex-centric operations on the global graph and the coprocessor is responsible for completing edge-centric operations on the local graph, capable of supporting graph computing on trillions of edges.

LUMOS [113] proposes a dependency-driven graph processing technique that actively propagates values between iterations through unordered execution while providing synchronous processing guarantees. The cross-iteration value propagation mechanism of LUMOS efficiently identifies future dependencies and can actively compute the values of dependencies without sacrificing disk locality, which can diminish the number of graph data that needs to be loaded in subsequent iterations and speed up graph processing.

In order to optimize the execution efficiency of graph computing tasks, DGraph [114] scales each strongly connected component of the graph into abstract graph vertices according to the dependencies between graph vertices. The graph is transformed into an abstract directed acyclic graph, and then the directed acyclic graph is divided into multiple layers so that there is no interdependence between the strongly connected components of each layer. Then, each strongly connected component is processed in parallel according to the level number so that a high number of graph vertices only need to be processed a few times to converge, greatly reducing the data access costs and redundant update times.

Wonderland [115] is a graph processing system based on graph abstraction, which can efficiently extract graph abstractions from raw graph data under specified memory constraints, and accelerate disjoint graph partitioning on disk through graph abstraction interval message propagation to improve graph processing performance. Wonderland also includes a priority scheduling strategy based on graph abstraction, which can effectively accelerate the convergence of graph algorithms.

Congra [116] and CongraPlus [117] explored the scheduling problem of concurrent graph computing requests, and designed a set of memory bandwidth-efficient single-machine graph computing scheduling strategies on the shared memory architecture. CongraPlus graph computation requests are implemented based on the Ligra framework. This technology obtains information about graph processing requests through offline sampling. In the running phase, CongraPlus first distributes requests to the local scheduler evenly through global request allocation, and then uses an iterative ascent algorithm to calculate the optimal number of threads required for each request. During this process, the system ensures that concurrently executed requests do not over-compete for memory bandwidth and uses the LookAhead algorithm to provide better performance for heavily loaded requests.

GraphM [101] is an effective storage system which can be easily embedded into the existing graph computing system and make full use of the data access similarity of concurrent graph computing tasks, allowing the graph structure data to be regularly flowed into memory/cache and shared by concurrent graph computing tasks, improving the throughput of concurrent graph computing tasks by reducing data access and storage overhead. Subsequently, GraphSO [118] adopts a fine-grained graph data management mechanism and uses an adaptive data repartitioning strategy and a structure-aware graph data caching mechanism at runtime to further reduce redundant I/O for concurrent graph computing tasks overhead and improve system throughput.

EGraph [102] is a GPU-based dynamic graph processing system that can be integrated into existing GPU out-of-core static graph processing systems and efficiently utilizes GPU resources to support concurrent processing of different snapshots of dynamic graphs. Unlike existing approaches, EGraph proposes an efficient Loading-Processing-Switching (LPS) execution model. It achieves efficient execution of temporal iterative graph processing tasks by making full use of the data access similarity between temporal iterative graph processing tasks to effectively reduce the CPU-GPU data transfer overhead and ensure higher GPU utilization.

3.1.2. Distributed Graph Processing Systems. A distributed graph computing system consists of multiple computing nodes, each of which has its own memory and external memory. Therefore, compared to single-machine graph computing systems, distributed graph processing systems are less limited by hardware in terms of scalability. However, in a distributed graph processing system, graph data is distributed to multiple nodes for processing. Therefore, the data partitioning mechanism has a great influence on the performance of the distributed graph processing system, and it is a challenge to design an appropriate data partitioning strategy. Meanwhile, the communication between computing nodes becomes a performance bottleneck, and the system's overall performance and the scale of data processing are limited by the network bandwidth.

Most distributed graph processing systems are distributed in-memory graph processing systems, in which all graph data is completely loaded into memory for processing.

Pregel [103] is one of the earliest distributed in-memory graph processing systems, which uses the batch synchronous parallel (BSP) model for processing graph data and proposes a vertex-centric computing framework that represents the graph algorithm as a series of iterations, where each vertex modifies its own state and the state of its output edges based on the messages it has received from previous iteration and sends the messages to other vertices. The vertex-centric computing framework is extremely expressive and can implement a large variety of graph algorithms.

Many distributed in-memory graph processing systems are extended with Pregel [103]. GraphLab [104] supports asynchronous execution of graph algorithms while ensuring data consistency, and this asynchronous model tends to have faster convergence and lower synchronization costs than synchronous models. However, GraphLab still suffers from vertex degree skew. GraphX [119] is a graph processing

framework on Apache Spark that combines the advantages of a dedicated graph processing system with those of a distributed data streaming system to provide a set of composable graph abstractions on a distributed data streaming system to implement and execute iterative graph algorithms efficiently.

To address the workload imbalance and other problems associated with power-law graphs, PowerGraph [105] adopts the Gather-Apply-Scatter (GAS) computation model to decompose the vertex program into multiple stages, allowing the computation to be more evenly distributed across the cluster. At the same time, vertex partitioning and a series of fast heuristics are employed to reduce the storage and communication overhead of power-law graphs on distributed clusters. PowerGraph can support the BSP computation model of Pregel [103] and the asynchronous computation model of GraphLab [104].

PowerSwitch [106] and PowerLyra [120] are designed based on the distributed graph computing system Powergraph [105]. Power-Switch proposes a hybrid computing model, which can automatically switch between asynchronous and synchronous execution modes during parallel graph computing processing to obtain the best performance. PowerLyra employs different graph division strategies and graph computation schemes for high-degree and low-degree vertices to improve system efficiency.

Gemini [121] is a computation-centric graph processing system, which applies the hybrid Push/Pull computing paradigm in distributed scenarios, and adopts a chunk-based graph partitioning strategy that exhibits good data locality at multiple levels of parallelism. Gemini builds a low-cost distributed design on the basis of optimizing the computing efficiency of a single node, which effectively improves the utilization of system resources.

Grape [122] is a graph computing system that can automatically convert serial algorithms into parallel algorithms. Its main benefit is that serial algorithms' logic does not need to be changed. This significantly lessens the challenge of parallel programming in graph computing since it can be done in parallel by putting it into Grape as a whole. In addition, since Grape divides vertices into a specified number of fragments, the vertices on each fragment can adopt a serial algorithm, and then parallelize the algorithm to support data partition parallelism so that optimization strategies developed for serial algorithms can be implemented in Grape. These optimizations are difficult to use directly in ordinary vertex-centric models.

To speed the convergence of graph computing tasks, Maiter [107] proposed the difference-based cumulative iterative computation (DAIC) theory, where each vertex only propagates and accumulates the updated value instead of the full value of the vertex, which can be executed asynchronously and efficiently. The graph iterative algorithm uses less iterative computation overhead to reach a convergent state, and rigorously proves the correctness of the DAIC computation.

To further accelerate asynchronous graph processing on distributed platforms, FBSGraph [123] proposes an efficient forward and backward scanning execution approach, which can greatly enhance the state propagation efficiency of asynchronous graph processing. FBSGraph also includes a static prioritization scheme, which can effectively reduce the communication overhead of asynchronous graph processing on distributed platforms, and further improve the convergence speed of asynchronous graph algorithms.

PowerLog [124] further explores the theoretical underpinnings of whether monotonic or nonmonotonic programs can be performed correctly asynchronously and incrementally and develops a conditional verification tool to automatically check whether the program satisfies this condition. Based on this, PowerLog designs a unified distributed synchronous and asynchronous engine. When executing recursive aggregation programs, it realizes adaptive asynchronous and synchronous execution modes by dynamically adjusting the frequency of message propagation to minimize program execution time. PowerLog's analytical approach is the basis of incremental computing for many graph processing and graph neural networks.

For efficient processing of streaming graphs, Kineograph [125] designs a distributed in-memory graph storage system to generate a reliable and consistent series of snapshots at regular intervals and uses a graph computation engine that supports incremental iterative propagation to process the snapshots to obtain real-time computation results. The core idea of real-time computation results is to interleave iterative computation and batch updates of graphs. Iterative computation maintains the intermediate computation results of the most recent version of the graph, and when a query is received, iterative computation is performed directly from the intermediate results to obtain the exact computation results of the current version of the graph after the batch update.

Tornado [126] optimizes real-time iterative graph processing on distributed platforms. It proposes an approximation method to enhance the timeliness of graph processing and designs a novel bounded asynchronous iterative processing model that can ensure the correctness of graph processing results and achieve efficient fine-grained updates.

During dynamic graph processing, changes in graph structure brought about by edge deletion may lead to invalidation or performance degradation of intermediate results of graph computation. For this reason, KickStarter [108] proposes a runtime technique to address the challenges posed by edge deletion. The key idea is to precisely identify the vertices affected by edge deletion based on inter-vertex dependencies and adjust the state values of these vertices to an estimate close to the convergence value. This runtime technique of tracking and adjusting vertex state values ensures that the computation yields correct results and accelerates the convergence of the graph algorithm.

Aiming at incremental computing for large-scale dynamic graphs, Ingress [109] can automatically realize the incrementalization of vertex-centric graph algorithms, and is equipped with four different computing state memory strategies. The strategy applies to the sufficient conditions of the graph algorithm, and the system can automatically select the optimal memory strategy according to the algorithm logic specified by the user.

Table 2: Overview of software systems for graph mining. (PA, GTS, MPO represent pattern-aware, graph-traversal strategy, multipattern optimization, respectively).

Year	System	Architecture	PA	GTS	MPO	Main features
2015	Arabesque [127]	Distributed CPUs	No	BFS	No	Embedding-centric model
2016	ScaleMine [128]	Distributed CPUs	No	BFS	No	Scalable and parallel
2018	G-Miner [129]	Distributed CPUs	No	BFS	No	Task pipeline
2018	RStream [130]	Single machine	No	BFS	No	GRAS model
2019	Fractal [131]	Distributed CPUs	No	DFS	No	DFS exploration
2020	G-thinker [132]	Distributed CPUs	No	BFS	No	Vertex caching Task scheduling
2020	Pangolin [133]	Single machine (CPU & GPU)	No	BFS	No	Efficient and flexible
2021	aDFS [134]	Distributed CPUs	No	Hybrid	No	Almost-DFS exploration
2019	AutoMine [135]	Single machine	Yes	DFS	Yes	Compilation techniques Matching order
2021	GraphZero [136]	Single machine	Yes	DFS	Yes	Symmetry order
2020	Peregrine [137]	Single machine	Yes	DFS	No	Matching order Symmetry order
2020	GraphPi [138]	Distributed CPUs	Yes	DFS	No	Optimal order
2020	DwarvesGraph [139]	Single machine	Yes	DFS	No	Pattern decomposition
2021	Kudu [140]	Distributed CPUs	Yes	Hybrid	No	Extendable embedding BFS-DFS exploration
2021	Sandslash [141]	Single machine	Yes	DFS	No	Two-level optimizations
2021	SumPA [142]	Single machine	Yes	DFS	Yes	Pattern abstraction
2022	G^2Miner [143]	Multiple GPUs	Yes	Hybrid	Yes	Input-aware Architecture-aware

The distributed out-of-core graph processing system expands the single-machine out-of-core graph processing system into a distributed cluster, thereby further expanding the scale of graph data. Based on X-Stream [97], Chaos [110] is currently the only distributed out-of-core graph processing system, and extends single-machine out-of-memory graph computation to multiple machines. It is composed of computing subsystems. Each machine has a storage subsystem as a storage engine, which provides vertices, edges, and updates for the computation subsystem. Chaos realizes parallel execution and sequential storage access by adjusting the X-Stream stream partition and achieves multimachine load balancing through random work stealing technology.

3.2. Software Graph Mining Systems. Recently, several software systems have been proposed to solve the graph mining problem. They search for subgraphs that satisfy the conditions of the algorithm in the input graph G. The process of finding subgraphs can be modeled with a search tree where each node represents a subgraph, and the subgraphs at the k+1 level are expanded from the subgraphs at the k level. Based on the model, these systems can be classified by programming model into two main types: pattern – oblivious and pattern – aware, also called embedding – centric and set – centric. They also adopt a variety of different techniques, such as graph-traversal strategy and multipattern optimization, to improve the performance of graph mining problems. Table 2 summarizes the software systems for graph mining.

3.2.1. Programming Model

(1) Pattern-Oblivious Programming Model. Pattern-oblivious systems [127–131, 133, 134] adopt the embedding-centric approach to solve the graph mining problem. They establish a search tree to represent the partial embeddings (nonleaf nodes) and final embeddings (leaf nodes). For the partial embeddings, some pruning techniques are employed to prevent duplication and unnecessary exploration. For the final embeddings, expensive isomorphic tests are applied to check if they are isomorphic to the pattern. Arabesque [127], the first distributed graph mining system, uses BFS to explore the search tree based on the BSP model, and proposes the coordination-free exploration strategy to avoid redundant exploration. However, Arabesque suffers from high memory and IO costs and synchronization overhead.

Memory and IO costs. Although BFS exploration provides high parallelism, the number of subgraphs increases exponentially with the pattern's size. When the input graph is large, keeping all subgraphs in memory is impractical, and employs out-of-core processing, which may cause an IO bottleneck. To keep CPU cores occupied while waiting for data, G-thinker [132] maintains a pool of active tasks that can be processed at any time. In this way, while some tasks are

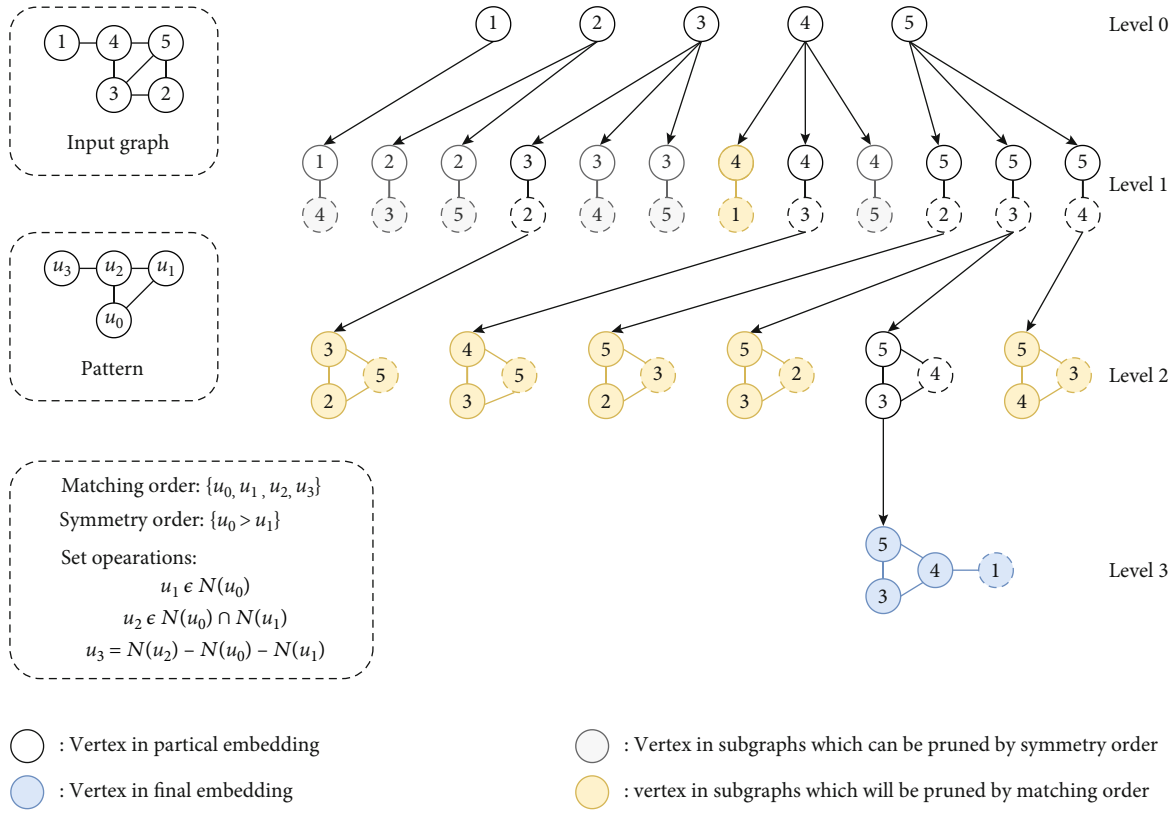

Figure 3: A four-level subgraph search tree for pattern-aware graph mining algorithm.

waiting for data, other tasks can continue their computations. Furthermore, it implements a novel vertex cache to support highly concurrent vertex accesses to minimize redundant requests. To reduce memory consumption, Fractal [131] adopts the DFS method to enumerate embeddings.

Synchronization overhead. The BSP model requires synchronization between supersteps, which leads to the straggler problem, lowering hardware utilization of distributed graph mining systems. To remove the synchronization barrier, G-Miner [129] proposes a novel task-pipeline design which can asynchronously process CPU computation, network communication, and disk I/O.

ScaleMine [128] introduces a novel two-phase solution to support the FSM algorithm in a single large graph with scalability and parallelism. In the first phase, it adopts an approximate approach to efficiently identify subgraphs with high probability. In the next phase, the exact solution is computed by utilizing the collected information from the first phase to ensure excellent load balancing and improve parallelism.

To overcome the drawbacks of conventional distributed mining systems, such as large startup and communication overhead and underutilization of CPU resources, RStream [130] develops the first single-machine, disk-based, out-of-core graph mining system. Based on relational algebra, RStream provides a rich programming model (named GRAS) to enable a variety of graph mining algorithms and efficient implementations of relational operations (particularly join) for graphs.

Existing mining systems, such as Arabesque and RStream, have limited performance because they are mainly focused on generality rather than application-specific customization and implementations of parallel operations and data structures. To solve the issues, Pangolin [133] provides high-level abstractions for the processing of graph mining on CPU and GPU and enables application-specific customization. Besides, efficient parallel operations and data structures are designed to improve hardware utilization.

(2) Pattern-Aware Programming Model. Existing pattern-oblivious graph mining systems made a lot of effort to solve several significant problems and achieved remarkable performance improvements. However, there are still issues with the costly overhead of subgraph isomorphism tests and pruning search space. To address the issues, the pattern-aware solution [135–138] analyzes the structure of the pattern and generates a matching order [137] and a symmetry order [136] to eliminate isomorphism tests and repetitive enumeration. The pattern-aware algorithm for mining a tailed triangle pattern with four vertices is shown in Figure 3.

Matching Order. Matching order is the search order of vertices in the pattern when performing graph mining. For example, in Figure 3, a matching order of the tailed triangle pattern is $\{u_0, u_1, u_2, u_3\}$, indicating that u_i is ancestor of u_j and searches prior to u_j only when $i > j$. The yellow colored subgraphs in Figure 3 can be pruned by the matching order, because the input graph does not contain the vertex that the

matching order wants to explore in the next step. For example, the subgraph $\{u_0, u_1\} = \{4, 1\}$ expends the next vertex u_2 of the pattern which is a common neighbor of u_0 and u_1. Since the intersection set of $N(4)$ and $N(1)$ in the input graph is empty, the subgraph $\{4, 1\}$ has no branches that meet the matching order and can be pruned. More importantly, isomorphism tests can be avoided using matching order, because the final embeddings always match pattern.

Symmetry Order. Although employing matching order can eliminate isomorphism tests and prune search space, a certain subgraph can be explored multiple times due to the symmetry of vertices in the pattern. As shown in Figure 3, u_0 and u_1 of the pattern are symmetric, and the subgraphs $\{2, 3\}$ and $\{3, 2\}$ are identical subgraphs which also called automorphisms. To eliminate duplication of enumerating these embeddings, symmetry breaking method establishes a symmetry order among the vertices of the pattern. For instance, in Figure 3, the symmetry order $u_0 > u_1$ is employed to prune the search space and ensure uniqueness. In particular, the grey colored subgraphs in Figure 3 such as the subgraph $\{2, 3\}$, which is automorphic to the subgraph $\{3, 2\}$, are pruned by the restriction of $u_0 > u_1$.

Optimal order. Previous pattern-aware systems adopted the matching order and symmetry order to avoid isomorphism tests and redundant enumeration. However, different matching orders and symmetry orders have an important influence on the performance of the graph mining system. To address the problem, GraphPi [138] first designs algorithms to generate multiple sets of matching order and symmetry order. Then, it discovers the optimal combination of matching order and symmetry order based on a precise performance prediction model. Furthermore, GraphPi leverages the technique of Inclusion-Exclusion Principle (IEP) to optimize the algorithms that just count the embeddings' number.

The execution time of embedding enumeration increases dramatically as the pattern size grows. To address the challenge, DwarvesGraph [139] develops a high-performance system using pattern decomposition techniques, which break down a large pattern into a number of smaller subpatterns, and then compute each of them separately. To support various applications, DwarvesGraph introduces a novel partial-embedding-centric programming model. To decrease memory consumption and random access, DwarvesGraph proposes an efficient on-the-fly aggregation of subpatterns embeddings. DwarvesGraph also designs a compiler that employs conventional and loop rewriting optimization and a novel lightweight cost model to estimate the performance of an algorithm implementation candidate.

Sandslash [141] provides a two-level (high- and low-level) programming model and corresponding optimizations, while earlier systems only focused on one level (i.e. either high-level [135, 137] or low-level [130, 133]). The high-level programming interface provides effective search strategies, data representations, and high-level optimizations such as matching order. The low-level programming interface allows the programmers to express algorithm-specific optimizations. Sandslash also flexibly explores combinations of optimizations to enhance performance.

3.2.2. Graph-Traversal Strategy. Exploration of the search tree generally follows one of two typical graph-traversal strategies: BFS or DFS, but different graph-traversal strategies have different parallelism and memory consumption. BFS explores the search tree level by level and maintains a list of intermediate subgraphs at each level that can be processed in parallel. Although BFS enables tremendous parallelism, it suffers from memory consumption due to the size of intermediate subgraphs grows exponentially. DFS reduces the size of intermediate subgraphs, but it is difficult to parallelize because of the data dependency, and has poor locality because of irregular memory access. To take advantage of the best of both strategies, many systems [134, 140, 143] employ a DFS-BFS hybrid strategy.

aDFS [134] proposes an almost DFS graph exploration strategy with a high degree of parallelism and constrained runtime memory consumption. In particular, threads in aDFS mainly prioritize DFS exploration and switch to BFS when waiting for the required data (e.g., edges on a remote machine) or when the exploration reveals low parallelism (e.g., a few intermediate subgraphs) that the runtime can identify.

The task granularity and execution schedule influence the efficiency of a distributed graph mining system with partitioned graph. To reduce task granularity and enable efficient scheduling, Kudu [140] proposes a well-defined abstraction of extendable embedding which has high expression for graph mining algorithms and enables fine-grained task scheduling and a BFS-DFS hybrid exploring approach which produces appropriate concurrent tasks with limited memory consumption, respectively. Specifically, the BFS-DFS hybrid exploration adopts DFS with a chunk granularity.

For problems that use domain support, such as FSM, G^2 Miner [143] proposes a bounded BFS search to fully utilize the GPU. It initially employs BFS search to generate massive parallelism, and then partitions the intermediate subgraphs into blocks, each of which can reside in memory when the intermediate subgraphs cannot fit in memory. After that, the subgraphs can be processed block by block. For the loading imbalance of DFS, G^2 Miner employs edge parallelism, which maps the subtree rooted at each edge at level 1 of the search tree to one task, instead of vertex parallelism (explanation similar to edge parallelism) to reduce task granularity and provide more parallelism.

3.2.3. Multipattern Optimization. Unlike a single-pattern problem, which only mines one single pattern at a time, a multipattern problem finds multiple patterns simultaneously. Running multipattern problems could result in redundant computations [142] and low hardware utilization [143]. To overcome these issues, graph mining systems such as AutoMine [135], G^2Miner [143], and SumPA [142] propose a variety of optimizations, the key insight of which is to merge multiple patterns that share the same subpattern to enjoy sharing.

AutoMine [135] first generates a schedule for each pattern, which is a sequence of set operations, and then combined those schedules to form the merged schedule.

Table 3: Overview of typical software systems for graph learning.

Year	System	Architecture	IM	Baselines	Main features
2019	DGL [144]	Single GPU	Yes	PyG	Compatible with multiple backends Message-passing parallelism
2019	PyG [145]	Single GPU	Yes	DGL	Optimization of sparse operations
2020	FeatGraph [146]	Single GPU	Yes	Gunrock	Optimization of matrix mult User-defined functions
2020	G^3 [147]	Single GPU	Yes	PyTorch TensorFlow	Using graph processing systems to support graph-structured operations
2021	GNNAdvisor [148]	Single GPU	Yes	DGL, PyG NeuGraph [149]	Profiling of GNN model and graph 2D workload management
2021	PyTorch-direct [150]	Multiple GPUs	No	PyTorch	Zero-copy programming model
2022	GNNLab [151]	Multiple GPUs	No	DGL, PyG	Factored space sharing design Presampling based caching policy
2021	P3 [152]	Distributed GPUs	Yes	DGL, ROC [153]	Eliminating high communication and partitioning overheads Pipelined push-pull parallelism
2021	DistGNN [154]	Distributed CPUs	Yes	DGL	Full-batch GNN training on CPUs mitigating communication bottlenecks
2022	ByteGNN [155]	Distributed CPUs	Yes	DGL, Euler	Two-level scheduling strategy New graph partitioning algorithm
2022	NeutronStar [156]	Distributed GPUs	Yes	DGL, ROC	Hybrid dependency management

Specifically, AutoMine analyzes prefixes of schedules, and overlapped prefixes can be shared to avoid repetitive enumeration and contribute to data reuse. The common search paths of schedules first begin to converge, and then diverge at some level k where the paths differ.

Unfortunately, the sharing the prefix technique in AutoMine only eliminates a small proportion of redundant computations, since the matching order must be the same even though prefixes of schedules construct the same subgraphs. SumPA [142] presents a pattern abstraction technique based on pattern similarity to guide pattern mining and eliminate (totally and partially) redundant computations. Specifically, SumPA proposes a redundancy criterion called Shared Connected Subpattern (SCS) to characterize redundant computation. According to the SCS similarity, it extracts a few sample abstract patterns from numerous complex patterns.

G^2Miner [143] employs a kernel fission technique to improve hardware utilization of mining multipatterns simultaneously on GPU. G^2Miner analyses multiple patterns to identify those patterns that share the same subpattern, and then they can share the same workflow by merging into the same CUDA kernel. In contrast, G^2Miner generates distinct kernels for those patterns that do not share the same subpatterns.

3.3. Software Graph Learning Systems. As one of the most popular research directions in the field of artificial intelligence in recent years, graph neural network has produced a large number of different algorithms. Many enterprises and research teams have carried out research and development work on the framework and extension library of graph-oriented neural network based on a common platform. Because graph neural network applications and traditional neural network applications have similarities and differences in many aspects of implementation methods, many existing works are expanded based on the mature neural network frameworks, such as PyTorch and Tensorflow, to form a new framework supporting graph neural network applications. These frameworks and extension libraries support many different graph neural network algorithms, most of which are open source, and it is convenient for users to construct graph neural network flexibly. The mainstream graph neural network framework and extension library will be introduced in the following. Table 3 gives an overview of typical software systems for graph learning.

3.3.1. Single-Machine Graph Neural Network System. A typical GNN system is divided into a data module and a computation module. Among them, the data module is mainly responsible for IO and preprocessing of data, and the computation module is mainly responsible for training and inference of algorithmic models. GNN systems with a single GPU are the first to receive attention in the early development of GNN acceleration systems. This is because in the case of relatively small graph structure and feature vector data, which can be stored directly in the GPU memory. Regarding this, a large amount of work has emerged, such as Deep Graph Learning (DGL) [144], PyTorch Geometric (PyG) [145], GNNAdvisor [148], and G^3 [147].

DGL [144] is one of the mainstream academic GNN programming frameworks currently integrated in mainstream neural network system architectures, such as PyTorch, Tensorflow, and MXNet. DGL follows the Message Passing Neural Network computational paradigm proposed by Google [157] to accomplish GNN training and inference. This computational paradigm contains three main

propagation phases, i.e., Message Passing Phase, Reduce Phase, and Update Phase, and its propagation formula can be expressed by the following equation ($y_{i'}$ denotes the output result of vertex i; y_i and y_j denote the input feature, respectively; $e_{i,j}$ denotes the feature of edge i to j; $\varnothing$ denotes the aggregation formula; γ_Θ denotes the activation function).

$$y_i' = \gamma_\Theta\left(y_{i,j\in N(i)} \varnothing\left(y_i, y_j, e_{i,j}\right)\right). \tag{1}$$

Message Passing Phase, i.e., for each edge on the graph, the message to be propagated for each edge is obtained by computing the features on the edge and the features of the two nodes on the edge.

$$\varnothing\left(y_i, y_j, e_{i,j}\right). \tag{2}$$

Reduce Phase is mainly computed for graph vertices, each vertex aggregates the messages on the edges by the reduce function, where the reduce function can be summation, maximum, minimum, and mean, etc.

$$_{j\in N(i)}. \tag{3}$$

Update Phase is also computed for graph vertices, and each vertex is updated by using the update function to update the feature vector of that vertex using the information of the previous layer of that vertex and the aggregated information.

$$\gamma_\Theta \text{ for each node.} \tag{4}$$

Moreover, DGL mainly adopts the graph-centric computation model, which means that the propagation computation and vertex/edge computation on the graph are implemented through the entire graph itself. To reduce the memory space occupied by the message tensor, DGL also employs message fusion techniques to merge multiple messages into a single message. However, DGL is based on the deep learning framework PyTorch and is developed with a graph manipulation module on top of it, which itself is less optimized for graph manipulation during GNN execution.

Similar to DGL, PyG [145] is also a GNN framework built on Python to model and train deep learning on graph-structured data. However, PyG provides optimization of sparse operations during GNN execution compared to DGL and employs a dedicated CUDA kernel to achieve efficient training. With PyG's defined message-passing interface, the user only needs to define the message and update functions and select the corresponding aggregation function functions, such as accumulation, maximum, and minimum, to define new GNN models.

Compared with the mainstream frameworks such as DGL and PyG, FeatGraph [146] is more concerned with GNN sparse operations, which is due to the fact that the GNN aggregation stage requires multiplying the graph feature vector matrix by the graph adjacency matrix. FeatGraph innovatively proposes the optimization of feature vector cuts and adopts a new graph partitioning method, which is a super sparse operation. FeatGraph also supports User-Definition Functions (UDFs) for different GNN models of edge aggregation and vertex aggregation functions for different GNN models. However, FeatGraph uses the full-batch training method because it needs to store all data in GPU memory.

PyTorch-direct [150] proposes a large-scale GNN framework based on zero-copy. The sparse node features stored in the CPU are the main reason for slow data loading in large-scale GNN training. Compared with explicit copy, which is suitable for transferring large blocks of contiguous memory, zero-copy is more suitable for dealing with random sparse access. However, the simple implementation of the zero-copy access model will bring the disadvantages of high latency and low bandwidth. Aiming at the shortcomings of zero-copy, it proposes a zero-copy programming model that ensures data access merging and alignment as much as possible.

Existing GNN acceleration systems use traditional deep learning frameworks as backends, such as DGL and PyG, and then add a graph engine module to support graph operations during GNN execution. However, G^3 [147] believes that the main reason for the inefficiency of the GNN execution process is the irregular traversal of graph topology and feature vectors in graph operations, so G^3 uses Gunrock [93] as the backend. It extends the existing automatic differentiation and neural network operators and other operations. Through Gunrock's own graph operation optimization technology, the efficiency of GNN training and inference is greatly improved.

GNNAdvisor [148] is a flexible and effective runtime system designed to accelerate GNNs on GPUs. It performs online analysis of the input graph and GNN operations to provide guidance to GPU-based workload and memory management agents. Specifically, GNNAdvisor first investigates and determines a number of performance-relevant characteristics from both the input graph and the GNN model. Then, GNNAdvisor provides an extremely effective 2D workload management to increase GPU performance and utilization in a variety of application scenarios. Finally, based on the community features of the graph, GNNAdvisor adopts a combination of node renumbering and reclassification algorithms to reduce memory access.

GNNLab [151] summarizes the minibatch training method based on graph sampling into three stages: sample, extract, and train. When using the GPU to accelerate the extract stage, the features of frequently accessed vertices are loaded into the GPU memory in advance to reduce the amount of feature data copied, thereby accelerating the extract stage. GNNLab designed a method based on space sharing, in which the training process is divided into two parts, with the Sampler in charge of the Sample stage and the Trainer in charge of the Extract and Train stages. In terms of cache strategy, GNNLab proposes a cache strategy based on presampling, which performs several rounds of graph sampling processes in advance, and then loads the features of the vertices with the highest number of samples during the presampling process into the GPU memory in advance.

3.3.2. Distributed Graph Neural Network System. The mainstream single-machine GPU GNN systems save all data in the GPU memory for GNN training and inference. However, in practical application scenarios, the graphs of GNN training and inference are huge. For graphs with a scale of billions or even tens of billions of vertices, it is impossible to store all data in the memory of a single node at one time. Therefore, it is critical to adopt a distributed GNN system to accelerate the efficiency of large-scale GNNs. At present, the typical distributed GNN systems mainly include DistGNN [154], Aligraph [158], and P3 [152].

Euler [159] uses the Tensorflow deep learning framework as the backend and extends support for the distributed CPU GNN framework. The Euler framework mainly includes three main modules: graph operation module, aggregation operator, and algorithm implementation module to rapidly expand the GNN model. Among them, the graph operation module is responsible for the storage of graph structure data and its feature vector data. The aggregation operator module supports a variety of aggregation operations on graph vertices or edges, such as globally weighted sampling vertices and edges. In the algorithm implementation module, Euler integrates a variety of common GNN algorithms, such as Scalable-GCN, a general algorithm for accelerating GNN tasks. However, Euler needs to cache the graph vertices and their K-hop neighbor vertices at one time.

P3 [152] is a distributed framework for large-scale GNN training. It focuses on reducing the network communication time and traffic of the vertex feature vector of the graph when using the minibatch training method. Different from traditional DNN distributed training, the graph structure used in GNN training is associative, and graph vertices or edges will have high-dimensional feature vectors. P3 innovatively proposes a pipeline-based push-pull training method, which stores graph structure data and eigenvector data separately, and divides the eigenvector matrix into columns. The column-wise segmentation of the eigenvectors means that for each node, there will be some dimensions of the complete eigenvector matrix for all vertices, and there is no need to pay attention to any graph distribution, so it does not need to be expensive to remote the machine node pulls the required feature vector data. Similarly, P3 also joins the pipeline parallel design, covering the network communication time.

DistGNN [154] is optimized based on DGL to support GNN training in full-batch mode efficiently on CPU clusters. DsitGNN optimizes DGL's aggregation primitives to speed up aggregation operations through chunked caching, dynamic thread scheduling, and optimized loop execution using LIBXSMM. To reduce traffic, DistGNN uses a vertex-cut based graph partitioning algorithm. At the same time, to further reduce the communication's impact on the training performance during the vertex aggregation operation on the graph, DistGNN introduces a delayed aggregation update strategy of cutting points, overlapping communication and calculation, and hiding the communication overhead during aggregation.

Dorylus [160] is a distributed GNN training framework that can scale to billion-edge graphs using low-cost resources. To solve the problem of limited memory of GPUs, which does not scale to large graphs, Dorylus leverages serverless computing to increase scalability, and the key idea is computational separation. Computational separation can enable a pipeline where computational and tensor-parallel tasks on the graph can completely overlap. It divides the training pipeline into a set of fine-grained training pipelines according to the type of data processed by the training pipeline.

Aligraph [158] is a comprehensive distributed GNN framework built on the Tensorflow deep learning framework, which can process very large-scale graphs. It mainly consists of three parts, namely, (1) storage, which implements partitioning using multiple algorithms depending on the characteristics of the graph to minimize data movement; (2) sampling, which allows for the definition of algorithms, and custom sampling of relevant node neighborhoods; (3) operators, which implements combination and aggregation functions.

In order to optimize data communication between multiple GPUs, the distributed graph communication library DGCL [161] is proposed mainly for efficient GNN training. At the core of DGCL is a communication planning algorithm tailored for GNN training, which collectively takes into account the full utilization of fast links, converged communication, avoidance of contention, and balancing the load on different links. Existing single-GPU GNN systems can be easily extended to distributed training using DGCL.

ByteGNN [155] is a GNN training system based on GraphLearn. ByteGNN analyzes the problems of existing distributed GNN training systems: higher network communication costs, lower CPU utilization, and GPUs that cannot bring significant training benefits. ByteGNN divides the minibatch of graph sampling into five basic operations and expresses the sampling process using a DAG composed of basic operations, which enables fine-grained parallelism within the sampling. ByteGNN designs a two-level scheduling strategy to adaptively adjust the resources allocated for sampling and training to maximize CPU utilization and speed up training. ByteGNN designs a partitioning method specific to minibatch graph sampling to reduce the large amount of network communication caused by graph sampling during training.

NeutronStar [156] proposed a large-scale graph neural network training system in a distributed heterogeneous environment supporting mixed dependency management. NeutronStar analyzes and summarizes the existing distributed GNN system from the vertex dependency management strategy and finds that there are two main methods. One is a method based on dependency caching. Specifically, for a k layer neural network, the vertices are divided into different subsets depending on the caching strategy. Each vertex subset (including vertex attributes and vertex labels) and its dependent k order neighbors will be assigned to a worker node for GNN training. The other is a method based on dependency communication, that is, instead of caching the training dependency subgraph, each vertex obtains the training dependency from the local or remote machine through communication. NeutronStar designs an automatic

Table 4: An overview of graph processing accelerators.

Year	Accelerator	Architecture	PM	EM	Generality	Baselines
2016	FPGP [22]	FPGA	V	Sync	BFS	GraphChi [96]
2017	ForeGraph [162]	FPGA	E	Sync	Various	FPGP
2018	AccuGraph [17]	FPGA	V	Sync	Various	ForeGraph
2017	Zhang et al. [31]	FPGA	V	Sync	BFS	FPGP
2018	Khoram et al. [33]	FPGA+HMC	V	Sync	BFS	Zhang et al. [31]
2021	ThunderGP [163]	FPGA	V	Sync	Various	Hitgraph [25]
2022	SPLAG [164]	FPGA	V	Async	SSSP	Hitgraph,ThunderGP
2016	Graphicionado [48]	ASIC	V	Sync	Various	GraphMat [165]
2018	Minnow [166]	ASIC	V	Async	Various	Galois [85]
2018	HATS [167]	ASIC	V	Sync	Various	IMP [168]
2019	GraphDynS [47]	ASIC	V	Sync	Various	Gunrock,Graphicionado
2020	GraphPulse [169]	ASIC	V	Async	Various	Ligra,Graphicionado
2021	DepGraph [44]	ASIC	V	Async	Various	HATS, Minnow, PHI [170]
2021	PolyGraph [52]	ASIC	V	Both	Various	Gunrock, GraphPulse, etc.
2015	Tesseract [78]	PIM	V	Sync	Various	DDR3-based system HMC-based system
2017	GraphPIM [79]	PIM	V	Sync	Various	GraphBIG [171]
2018	GraphP [82]	PIM	V	Sync	Various	Tesseract
2018	GraphR [69]	PIM	V	Sync	Various	GridGraph,Gunrock,etc.
2019	GraphQ [83]	PIM	V	Both	Various	Tesseract
2020	GaaS-X [172]	PIM	V	Sync	Various	GraphR

dependency management module, which selects the processing strategy with the least cost for each vertex, to maximize the utilization of communication and computing resources.

4. Domain-Specific Architectures for Graph Analytics

4.1. Graph Processing Accelerators. Graph processing is widely used to analyze complex relationships among entities in fields such as machine learning, roadmap, and social networks. With the era of big data, real-world graphs have become even larger and more complex. Due to the graph's irregular characteristic, traditional architectures, such as CPUs and GPUs, have difficulty completing graph processing with high performance and low energy consumption. Therefore, architecture innovations in graph processing are urgently needed to support the increasing scale of graph processing. Graph processing accelerators have been designed on various hardware platforms, including FPGAs, ASICs, and PIMs. Accelerators designed on FPGAs and ASICs typically feature dedicated computational logic circuits and memory hierarchies to accommodate the irregular computations present in graph processing. In situ computation without excessive data movement is emerging to accelerate memory-bound applications such as graph processing. Novel memory devices, such as HMC and ReRAM, are used to build PIM-enabled accelerators to accelerate graph processing. Next, we will review the aforementioned graph processing accelerators. Table 4 gives an overview of graph processing accelerators.

4.1.1. FPGA-Based Graph Processing Accelerators. With the characteristics of flexibility and low energy consumption, FPGA is widely used in boosting the performance of graph processing. With the limited on-chip resources, the key to designing the FPGA-based graph processing accelerators is efficiently utilizing the on-chip memory resources, effectively handling the conflicts of the pipelines and improving the utilization of the off-chip memory bandwidth.

Existing Efforts on FPGAs. The high bandwidth of block RAMs (BRAMs) on FPGA enables high-throughput random data access, which can effectively alleviate the high bandwidth requirement in graph processing. However, the memory of BRAMs is too finite to store the whole graph data. Therefore, many existing works extend to improve the utilization of BRAM. By partitioning the fine-grained graph data, both FPGP [22] and ForeGraph [162] improve the multiplex efficiency of the on-chip data, reducing the memory access latency. The overall framework of FPGA-based graph processing accelerator is shown in Figure 4. Shared vertex memory can be accessed by all FPGA chips. Each FPGA chip contains the Processing Kernels, which provide programmable logic enabling designers to create graph updating kernel functions, and on-chip caches for read and write of vertices.

When a high-degree vertex is frequently accessed, the pipelines will probably be blocked due to the atomic protection for the vertex updates. AccuGraph [17] fills this gap. By executing atomic operations in parallel through a specific accumulator, AccuGraph [17] eliminates the serialization of the conflicting vertices updates in atomic protection, reducing the conflict waiting in pipeline.

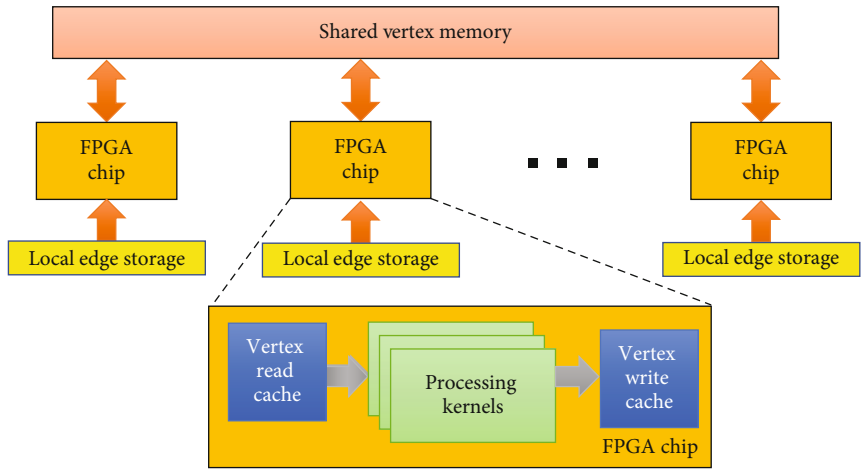

Figure 4: FPGA-based graph processing accelerator framework.

A number of studies focus on improving the utilization of the off-chip storage bandwidth. FabGraph [23] provides a two-level vertex caching technique. The UltraRAM serves as the L2 cache to interact with the DRAM, while the BRAM serves as the L1 cache coupled to the pipelines. To conserve DRAM bandwidth during processing, the L1 cache interacts with the L2 cache.

Integration with Hybrid Memory Cubes. With the development of technology, emerging storage hardware also provides us new solutions to solve the challenges in graph processing, e.g., HMC. Utilizing the low latency in random access and the high parallelism of HMC, Zhang et al. [31] achieved more than an order of magnitude throughput improvement on BFS compared to FPGP. Furthermore, Khoram et al. [33] analyzes the previous data access patterns of HMC and observes that the flag read operation contributes the most to the HMC access frequency, which severely affects performance. By fine-grained clustering and merging of memory requests, Khoram et al. [33] solves the performance bottleneck of previous work, resulting in improved performance and scalability.

Emerging Efforts on FPGAs. Recently, there are still new researches about the FPGA-based graph processing accelerator. GraSu [173] develops a library for FPGA to leverage the spatial similarity for quick graph updates, aiming to close the gap that current FPGA-based graph accelerators can hardly handle dynamic graphs. In order to lower the threshold of hardware design and improve programmability, ThunderGP [163] provides convenience for developers. With no knowledge of the hardware, the developers can still easily design by writing only a few high-level functions. Besides, SPLAG [164] accelerates SSSP for power-law graphs by using a coarse-grained priority queue (CGPQ) to enable high-throughput graph traversal. In conclusion, recent work on FPGA-based graph accelerators presents a diversified trend.

4.1.2. ASIC-Based Graph Processing Accelerators. ASIC is an integrated circuit chip that is customized for a particular use. It is composed of electrical components such as resistors. Typically, ASIC chips are fabricated on electronics wafers using metal-oxide-semiconductor technology. Different from the FPGA whose hardware logic can be reprogrammed to implement different functions, the function of an ASIC chip is hard-wired at the time of manufacture and has no chance of being changed. ASICs have the advantages of compaction, high speed, and low power, which introduce substantial opportunities to the design of graph processing accelerators.

Memory Hierarchy Designs. The irregular memory access pattern is a key characteristic and bottleneck of graph workloads. There are several graph processing accelerators proposed to optimize the memory latency- or bandwidth-bound problem. Graphicionado [48] proposes a hardware pipeline for graph processing that focuses on the datatype and memory subsystem. Specifically, the frequently accessed vertices property data is kept in a sizable on-chip eDRAM scratchpad memory. The on-chip memory can significantly improve the throughput of random vertex accesses compared with the expensive off-chip memory access.

Except for using a large on-chip scratchpad memory to optimize irregular memory access, fully exploiting data locality is another important optimization method. Prefetching is a popular method to optimize locality. Minnow [166] is an accelerator that offloads worklist scheduling and performs prefetching based on the worklist. It exploits knowledge from upcoming tasks to operate accurate prefetching in real-time. HATS [167] proposes and implements bounded depth-first scheduling (BDFS), which is a straightforward yet very efficient online schedule method to enhance the locality of graph applications. BDFS-HATS can significantly reduce memory accesses with inexpensive extra hardware overhead. GraphDynS [47] tries to deal with irregularities from the data-dependent graph algorithm behavior which is the origin of irregularities. It features a hardware/software cosdesign with separated datapath and data-aware dynamic scheduling. GraphDynS proposes a new programming model to extract data dependencies on-the-fly and presents several dynamic schedule strategies based on data dependency information. As a result, GraphDynS can significantly alleviate the irregularity and achieve higher performance and lower energy consumption than previous works.

Computing Units Designs. ASIC-based accelerators introduce opportunities to design and implement more efficient computing units to achieve higher graph processing performance. Designing novel processing frameworks with different program driving methods is generally adopted in [44, 169]. GraphPulse [169] is an asynchronous graph processing hardware framework with event-driven scheduling. By applying optimization strategies such as coalescing events and prefetching, GraphPulse can achieve high throughput. In addition, the asynchronous processing model brings more opportunities for task scheduling. DepGraph [44] proposes a dependency-driven asynchronous execution approach. Through the dependency chains which are generated efficiently at runtime, DepGraph can accelerate the propagation of vertices' states, thereby helping speed up the algorithm convergence.

Flexibility Oriented Designs. The aforementioned accelerators are focused on a certain graph algorithm variant. Different graph workloads and input graph datasets are performed well on different accelerators designed with different optimization strategies. None of the aforementioned accelerators can get the best performance for all scenarios. PolyGraph [52] explores the relationship among graph workloads, graph datasets, optimization strategies, and final performance. PolyGraph classifies algorithm variants from four dimensions including update visibility, vertex scheduling, slicing scheduling, and update direction and further proposes a heuristic method to specify which variant is best for a given workload and dataset. By proposing the execution model called "Taskflow," which integrates dynamic task parallelism and pipelined dataflow execution, PolyGraph implements a unified accelerator supporting different algorithm variants efficiently. Due to the exposed flexibility of graph accelerators, PolyGraph can achieve better performance on a variety of algorithm variants than previous accelerators.

Productivity Oriented Designs. Programmability is also a key issue in ASIC-based graph accelerator design. Seasoned designers usually use hardware description languages such as Verilog and VHDL to develop ASIC-based accelerators. However, prototyping an accelerator from scratch is challenging and error-prone due to the sizable design efforts involved. Many previous software-based works have addressed the programmability problems by proposing graph processing programming frameworks that handle system-related complexities including work scheduling, data communication, synchronization, and reliability, while domain experts only need to provide the application-level data structures and operations to develop distributed parallel programs. Ozdal et al. [51, 174] propose an architecture template and implement it as synthesizable SystemC models. For application developers, it is easy to specify data structures and operations related to applications to generate different complex graph processing hardware accelerators. In addition, by proposing some microarchitectural features such as allowing processing hundreds of vertices/edges simultaneously to conceal high memory access latency, the template-based graph accelerator can achieve high throughput and work-efficiency for graph applications. Moreover, it uses hardware primitives to address potential hazards and guarantee the correctness of final results.

4.1.3. PIM-Based Graph Processing Accelerators. Graph processing accelerators based on traditional architectures exhibit remarkable performance. However, the extremely low ratio of computations to data movements causes data to be moved frequently between compute and storage units, incurring significant time and energy overheads. With the emergence of PIM devices [79, 175], which incorporate the processing unit in the memory, the efficiency of data processing and transferring are significantly enhanced. As aforementioned, current PIM-based graph processing accelerators can be divided into PNM and PUM.

Considering the irregular memory accesses in graph processing, the PNM-based architecture integrates the computing logic inside the memory chip to reduce the data movement. Tesseract [78] is the first full-featured large-scale graph accelerator based on PNM, which follows the programming model and message passing mechanism of the distributed CPU graph processing system GraphLab [104] to ensure usability. Due to the inefficient interconnection interactions between HMCs, Tesseract suffers from communication blocking, resulting in large communication overhead. Based on Tesseract, GraphP [82] is proposed, which adopts an efficient interconnection network and reduces the number of communications between HMCs by optimizing the data partitioning. In contrast to reducing the number of communications, GraphQ [83] focuses on the overhead of a single communication. It packages messages and sends them in a nonblocking manner to reduce the overhead of a single message communication. Unlike previous works, GraphPIM [79] proposes a CPU-HMC heterogeneous graph processing architecture. Observing that vertex updates involve a large number of atomic operations, GraphPIM offloads the atomic operations to the HMC, reducing data access overhead and thus improving overall performance.

PUM is another type of PIM architecture that exploits the analog properties of the memory cell itself to perform computational functions. It allows the memory cell to store data and perform in situ calculations at the same time, essentially eliminating the movement of data during the calculation process. Considering that the calculation operations for graph processing are typically simple, only minor circuit modifications to existing memory can meet the computational requirements.

ReRAM is representative of this type of PUM architecture which could be the essential hardware foundation for performing matrix-vector multiplication in graph processing [69]. GraphR [69] adapts the graph algorithm to the matrix-vector multiplication paradigm and achieves significant performance gains while maintaining the correctness of the algorithm. Since real-world graph data is usually extremely sparse, the ReRAM-based accelerator suffers from a large number of invalid computations during computation. Spara [74] reorganizes highly correlated data together by exploiting the intrinsic community properties of graph data, thus improving the execution efficiency of the ReRAM crossbar

Table 5: An overview of graph mining accelerators.

Year	Accelerator	Architecture	Memory design	Computing design	Flexibility
2020	Gramer [50]	FPGA	Hybrid cache	Pipeline	
2021	FAST [176]	FPGA	Partitioned CST	Pipeline Task parallelism	CPU-FPGA codesign
2020	TrieJax [41]	ASIC	Dedicated cache	Join optimization	Coprocessor of CPU
2021	FlexMiner [54]	ASIC	C-map	Multicore Set operations	Software/hardware interface
2022	FINGERS [43]	ASIC		Branch parallelism Set parallelism Segment parallelism	Software/hardware interface
2022	SparseCore [177]	ASIC	Hybrid cache	Stream computing	ISA Sparse computing
2021	SISA [71]	PIM	Hybrid data representation	Hybrid computing	ISA Graph learning
2022	NDMiner [178]	PIM	Load elision Data parallelism	Composite computing	ISA
2022	DIMMining [179]	PIM		Index precomparison BCSR format Systolic merge array	ISA

arrays during practical execution. Considering the write cost is higher than read cost, GaaS-X [172] stores graph data in a sparse form in the ReRAM to avoid redundant writing of data.

4.2. Graph Mining Accelerators. Graph mining applications aim to find user-interested subgraph patterns, e.g., subgraph matching, motif counting, and frequent subgraph mining. Graph mining algorithms are usually NP-hard and consume huge amounts of computational resources and memory space due to the large search space of subgraphs. Mining simple size-4 cliques in a small graph with only 1 million edges can take several hours to finish on an 8-node cluster [137]. The size of generated intermediate data can easily reach tera-bytes for graphs with million-level edges. In real-world scenarios, the scale of graph mining problems is much larger, requiring numerous resources to finish. It is urgent to seek help from hardware approaches to accelerate graph mining problems. Early proposed graph processing accelerators have presented promising potentials for accelerating graph applications with emerging hardware. However, these accelerators cannot be directly used for graph mining because the underlying processing paradigms and key challenges are largely different. Graph mining accelerators explore various ways to exploit the potential of hardware to accelerate graph mining applications, from specialized algorithm mapping to generalized instruction-level optimizations. Generally, according to the oriented hardware platforms, the graph mining accelerators can be divided into three categories, i.e., the FPGA, ASIC, and PIM-based accelerators. Table 5 gives an overview of graph processing accelerators.

4.2.1. FPGA-Based Graph Mining Accelerators. Graph mining applications suffer from numerous random memory accesses to vertices and edges. FPGA offers fast on-chip memory such as BRAM which is suitable for accelerating the graph mining operations. Existing works mainly focus on cache hierarchy designs to facilitate the FPGA.

GRAMER [50] is the world's first general-purpose domain-specific graph mining accelerator. GRAMER employs the embedding-centric programming model that continuously extends and traverses all subgraphs to find interesting graph patterns. This model requires many random accesses to both vertices and edges of an embedding during the expansion phase. In graph mining, a tiny proportion of vertex and edge data results in the majority of memory accesses. The power-law characteristic of real graph structure and the scaling feature of graph mining itself provide the fundamental rationale. In the procedure of expanding the embedding in graph mining, high-degree vertices are more likely to be accessed. When the embeddings are expanded again, these newly added high-degree vertices will be accessed frequently, resulting in an obvious power-law distribution of data access. The percentage of accesses to the first 5% of the edge data will exceed 90% in the subsequent iterations.

To accurately locate and efficiently cache these edges to exploit their locality, GRAMER proposes a locality-aware on-chip cache design, which consists of eight independent cache partitions to handle different access requests simultaneously. For the 5% of frequently accessed vertex and edge data, GRAMER uses a high-priority cache implemented as scratchpad memory for static storage, thus minimizing the access latency of frequently accessed data. For the remaining 95% of the vertex and edge data, GRAMER uses a low-priority dynamically managed cache, further improving the overall access performance.

FAST [176] is the first CPU-FPGA heterogeneous accelerator dedicated for subgraph matching. Implementing subgraph matching on FPGA is nontrivial work, facing the challenges of strictly pipelined design and limited on-chip

memory resources. FAST solve the challenges with a novel software/hardware codesign that flexibly schedules tasks between CPU and FPGA to fully unleash the power of these two devices. In FAST, the CPU is responsible for scheduling tasks and maintaining an auxiliary Candidate Search Tree (CST) to be processed by FPGA. The CST is partitioned so that each partition can fit the limited FPGA resources. On the FPGA side, FAST proposes a three steps subgraph matching algorithm to support massive parallelisms for pipeline execution. Specifically, based on the matching order, a generator extends partial results and finds new subgraphs, then a Validator component checks the validation condition of newly generated intermediates, after which a Synchronizer fetches and reports the results. Each step can handle many partial results at a time so that the FPGA pipeline is fully utilized. FAST also proposes a BRAM-only matching mechanism to cache the partial results to avoid expensive external memory access. Leveraging the main memory of CPU-FPGA codesign architecture, FAST can easily support billion-scale graphs.

4.2.2. ASIC-Based Graph Mining Accelerators. Current ASIC-based graph mining accelerators all use the pattern-aware methodology to conduct graph mining applications, for the bounded memory footprint and inherent high concurrency. Existing research all adopt software/hardware codesigns to mapping graph mining applications to hardware logic, from algorithm-level mapping [41] to instruction level extensions [177].

TrieJax [41] is an energy-efficient on-die accelerator specialized for graph pattern matching problems. The key innovation of the accelerator design originates from a variant of the advanced Worst-Case Optimal Join (WCOJ) algorithms, Cached TrieJoin (CTJ), which is highly amenable to hardware designs. WCOJ algorithms naturally bound the intermediate results, which are friendly to limited hardware resources and provide inherent concurrency for specialized hardware parallelization. In order to support the CTJ algorithm, TrieJax architects the accelerator as a coprocessor that can be integrated into traditional CPUs as an additional core. TrieJax can directly access the graph in the main memory in an efficient way while leaving the scarce on-chip scratchpad memory for caching the intermediate results. Specifically, TrieJax decouples the CTJ algorithm into normalized join (TrieJoin) and cached join workflows by a SRAM-based partial join results cache. The TrieJoin module extends intermediate subgraphs by conducting basic join operations. The computed results are scheduled by a Cupid component in the cached join workflow. The recurring computations can reuse the cached results. TrieJax also enables multithreading to further hide memory latency and pipeline the cache reusing procedure.

FlexMiner [54] is the first general-purpose pattern-aware graph mining accelerator that incorporates a software/hardware codesign. Graph mining problems can be transformed into a series of graph pattern matching tasks, that is, a set of graph patterns to be explored. FlexMiner decouples the graph mining application development from the hardware design using a flexible compiler that automatically transforms the user input patterns into hardware execution plans. The execution plan defines the exact matching order of graph pattern vertices and corresponding operands and basic operators for each iteration. Given a series of patterns, the execution plans are represented by a specialized intermediate representation (IR), which supports multipattern optimization and provides hints for data management. The underlying hardware architecture of FlexMiner implements an efficient DFS tree walker that can traverse the entire subgraph search space to find unique subgraphs. A number of processing elements (PEs) are connected via an on-chip network. These PEs are independent of each other, each is composed of an extender, a pruner, and a reducer component. The extender adds new vertices to an intermediate subgraph, and then the pruner validates the candidates by a c-map data cache and the SIU/SDU units conducting set intersection/difference. The reducer finally operates the user-defined functions to compute statistics for each pattern according to the explored subgraphs. The software/hardware codesign helps FlexMiner to achieve high performance while maintaining ease of programming.

FINGERS [43] is an accelerator that fully exploits the multilevel fine-grained parallelism in the procedure of pattern-aware graph mining. Previous graph mining accelerators usually adopt a DFS mode to traverse the subgraph search tree to save memory footprint and only consider the coarse-grained parallelism, e.g., parallelizing the starting vertices or staying on the subgraph level parallelism. However, during the exploration of subgraphs, there is still parallelism to be exploited at the branch, set, and segment levels. Specifically, the branch-level parallelism is exposed by losing the strict DFS mode to traverse the branches of the search tree in parallel. The set-level parallelism is utilized by conducting multiple set operations simultaneously to exploit data locality. The segment-level parallelism is inside a set operation, and the data of a set is divided and processed in parallel. FINGERS implements a similar high-level architecture as in FlexMiner and inherits the benefits of easy programming. For PE designs, FINGERS enhances the processing logic to support multilevel parallelism and proposes a novel scheduling strategy and data organization to fully utilize the computation resources.

There are also works to explore instruction set architecture (ISA) and corresponding hardware designs specialized for graph mining. SparseCore [177, 180] propose the stream instruction set extension which represents the core computational operations in graph mining. Specifically, a stream is defined as a sparse vector in SparseCore. The stream is expressive and can represent various data formats, e.g., the edges of a graph and a list of (key, value) tuples. Stream ISA provides multiple extensions to traditional ISA, i.e., initializing and freeing a stream, computing on streams, and accessing the stream. Specifically, the operands of the stream instructions are general registers denoting which streams are involved. SparseCore supports three types of basic computations on streams, intersection, subtraction, and merging. The output of these computations can be further defined if only counting is needed. According to the stream ISA, SparseCore also presents a hardware design using the same

conventional processor architecture. There are multiple parallel stream units that process the computations. The architecture is optimized by exploiting the data reuse of streams with a stream cache and coordinating among streaming units to improve sparse computation. The stream ISA-based design provides the flexibility of supporting complex sparse applications, such as graph pattern mining algorithms and learning applications.

4.2.3. PIM-Based Graph Mining Accelerators. Graph mining can be represented in a series of set operations that are memory bound. The high bandwidth of PIM architecture naturally suits graph mining applications. Existing PIM-based graph mining accelerators exploit the inherent data parallelism from both the set operations and the memory architectures to maximize performance.

SISA [71] provides a set-centric ISA to support graph mining on PIM architectures. Specifically, vertices and edges can be represented as sets. SISA supports high-throughput set intersection and set union, while these set operations usually consume most of the runtime in existing CPU systems. The reason mainly lies in the high bandwidth requirement when conducting these operations because the edges as the operands are frequently accessed, which is proved in the stalled CPU cycle analysis in SISA. PIM architectures naturally fill in the gap between the bandwidth requirement and what memory devices can provide. However, the large amounts of intermediate results, interdependencies, and workload imbalance issues of graph mining make it challenging to utilize PIM. SISA tackles these challenges by designing an ISA based on set algebra and exploiting various possible PIM solutions to parallelize the set computations flexibly. Specifically, SISA supports several high-performance set intersections, set unions, and set difference operations. These instructions are optimized considering both the sparse and dense sets. A set can be stored as a sparse array of integers or dense bitvectors. The bitvectors is processed using in situ PIM methods (SISA-PUM) that directly compute through the DRAM circuitry with a little modification on the DRAM rows. There are no changes to the DRAM interfaces. The sparse array is processed in a near-memory mode (SISA-PNM) like the HMC architecture. These two computation modes are selected on-the-fly to take the best of both. SISA also provides high-level programming interfaces. Users only need to invoke an opaque type set and program in the normal way.

NDMiner [178] adopts the set-centric model and leverages the Near Data Processing (NDP) idea to tackle memory inefficiencies in graph pattern workloads. Through a thorough analysis of graph pattern mining, NDMiner characterizes the memory inefficiencies from four aspects. First, the operands of a set operation are usually distributed in different DRAM banks. Second, existing systems suffer from bandwidth wasting and low cache efficiency caused by symmetry-breaking optimization. Third, recurring set operations result in the same data being redundantly read. Fourth, the limitation on the capacity of the memory controller prevents graph pattern mining applications from fully using the internal data parallelism of the DRAM. To solve these inefficiencies, NDMiner exploits multilevel (i.e., bank, rank, and channel-level) parallelism in a DIMM-based DRAM and proposes novel architectural extensions specialized for graph mining. Specifically, for improving the utilization of bandwidth, NDMiner devises a load elision unit to abort unnecessary workloads brought by symmetry breaking. In order to remove the redundancies between different operations, NDMiner integrates compiler optimizations to merge possible computations and enables data reuse. It also proposes a reordering approach that dedicatedly maps graph data to the DRAM and schedules the set operations accordingly to fully utilize the multilevel data parallelism. For programming, NDMiner provides an ISA extension and modifies the memory controller accordingly.

DIMMining [179] is a DIMM-based PIM accelerator that solves the bottleneck of heavy comparison operations and data transfer for parallel graph mining. DIMMining adopts the principle of software/hardware codesign. Specifically, for alleviating the cost of pruning during runtime, DIMMining proposes an index precomparison strategy that partitions the neighbors of targeting vertices into small sets and identifies only necessary parts of the neighbors for comparison. According to the index precomparison, the comparison operations during runtime are largely reduced. To explore high parallelism in core set operations, i.e., set intersection and subtraction, DIMMining designs the BCSR data structure that stores the neighbor sets in a compressed bitmap format which is friendly to memory capacity and provides high parallelism. The set operations work on the BCSR format and are executed under a novel systolic merge array (SMA) structure which offers high throughput. DIMMining is architected with rank-level near memory computing processors in Load-Reduced DIMM. Two computing modules are added to the rank while the registering clock driver is modified to achieve instruction decoding and selection of computation modes. DIMMining can also switch between memory mode and computing mode.

4.3. Graph Learning Accelerators. As graph structured data is increasingly employed in various scenarios, emerging graph analytic applications, represented by graph convolutional networks (GCNs), have been widely pervasive. In contrast to traditional graph analytic applications, GCNs exhibit an incredible complexity. On the one hand, from the data perspective, the attributes of vertices and edges within the graph are typically more diverse instead of a single scalar information. On the other hand, in terms of computation, GCNs incorporate not only the memory-intensive graph traversal operations in traditional graph analytic applications, but also the computation-intensive vertex feature extraction operations in neural network applications, exhibiting heterogeneous computational characteristics. This complexity renders existing architectures failing to implement GCNs efficiently, thus several research efforts attempt to customize hardware units to accelerate GCN execution. Table 6 gives an overview of graph learning accelerators.

4.3.1. FPGA-Based Graph Learning Accelerators. FPGAs allow for the implementation of various circuits by

Table 6: An overview of graph learning accelerators.

Year	Accelerator	Architecture	Data layout	Edge scale	Preprocessing	Scheduling	Generality
2020	GraphACT [37]	FPGA	CUST	$[10^5, 10^7]$	Yes	Agg ⟶ Com	GCN
2020	AWB-GCN [181]	FPGA	CSC	$[10^3, 10^8]$	No	Com ⟶ Agg	GCN
2021	I-GCN [182]	FPGA	CUST	$[10^3, 10^8]$	No	Com ⟶ Agg	GCN
2021	ACE-GCN [183]	FPGA	CUST	$[10^3, 10^6]$	No	Com ⟶ Agg	GCN
2020	HyGCN [49]	ASIC	CSC	$[10^3, 10^8]$	Yes	Agg ⟶ Com	GCN
2020	Auten et al. [184]	ASIC	CUST	$[10^3, 10^4]$	No	Com ⟶ Agg	GNN
2021	EnGN [42]	ASIC	CUST	$[10^4, 10^8]$	Yes	Agg ⟶ Com	GNN
2020	GRIP [185]	ASIC	CUST	$[10^6, 10^7]$	Yes	Agg ⟶ Com	GNN
2022	SmartSAGE [186]	PIM	CSR	$[10^9, 10^{10}]$	Yes	Agg ⟶ Com	GCN
2022	HolisticGNN [187]	PIM	CSC/CUST	$[10^3, 10^7]$	Yes	Agg ⟶ Com	GNN
2021	PIM-GCN [72]	PIM	CSC/CSR	$[10^3, 10^8]$	Yes	Agg ⟶ Com	GCN
2021	TARe [75]	PIM	CUST	$[10^3, 10^7]$	Yes	Flexible	GCN
2022	ReFlip [3]	PIM	CSC/CSR	$[10^7, 10^8]$	Yes	Flexible	GCN
2021	DARe [188]	PIM	CUST	$[10^5, 10^7]$	Yes	Agg ⟶ Com	GNN

organizing the available hardware resources and programming them through low-level hardware description languages (e.g., Verilog), offering high parallelism and customizability. Therefore, FPGAs are widely used for GCNs acceleration, which can effectively alleviate the execution inefficiency of GCNs in general-purpose architectures. [37, 181, 182].

GraphACT. GraphACT [37] proposes a heterogeneous CPU-FPGA accelerator for GCNs training. GraphACT selects a subgraph-based minibatch algorithm [189] from various GCN training algorithms to minimize CPU-FPGA communication costs. According to the different hardware characteristics of CPU and FPGA, the authors partition the workloads between FPGA and CPU. CPU performs communication-intensive operations, including preprocessing, graph sampling, and nonlinear functions. FPGA executes the key steps of GCN training, such as forward and backward propagation passes. Moreover, GraphACT presents optimizations for the scheduling of FPGA modules and the scheduling between CPU and FPGA, which improves the execution efficiency of the pipeline.

AWB-GCN. GCN inference applications struggle to improve performance when processing large-scale graph data with highly unbalanced nonzero data distributions and extremely high sparsity. To address the issue, AWB-GCN [181] proposes three autotuning techniques to balance the workload. Specifically, the first one is called distribution smoothing to balance the workload between neighboring processing engines. The second one is called remote switching to exchange the workloads between busy and idle processing engines. The third one is called evil row remapping, which partitions and distributes the workloads of the evil row (containing too many nonzero elements) to a set of under-overloaded processing engines. AWB-GCN also explores the parallelism of intralayer and interlayer computation, with the benefit of reducing the overall latency and avoiding pipeline bubbles.

I-GCN. This work proposes an FPGA accelerator with an online graph restructuring algorithm for GCN Inference. The authors [182] argue that the cost of the offline preprocessing method previously proposed to solve the problems of poor data localization and redundant computation in the GCN inference process cannot be ignored since real-world graphs are frequently dynamic. I-GCN provides an algorithm called Islandization, which can identify internally connected vertex groups (Islands) and nodes with high degrees (hubs) in the graph at runtime. When processing each island, only the associated data of the internal island nodes need to be accessed, significantly reducing the communication between the accelerator and the external memory. Furthermore, the aggregation phase can identify and eliminate redundant computations because of the large number of common neighbors between nodes inside the islands.

ACE-GCN. ACE-GCN [183] proposes a data-driven FPGA accelerator, which exploits the inherent high sparsity and power law distribution commonly exhibited by real-world graph datasets. It provides the implicit-processing-by-association concept, which is similar to Islandization in I-GCN to guide the design of the accelerator. Specifically, based on estimation of graph structural similarity, ACE-GCN is able to automatically provide faster and less-expensive embedding estimations, which finally reduces the neural network workload and accelerates inference. In addition, DRAGON [190] expands ACE-GCN to support dynamic graphs.

4.3.2. ASIC-Based Graph Learning Accelerators. In this subsection, we focus on ASIC-based GCNs accelerators. Considering irregular data accesses and dense neural computations, several specialized ASIC-based accelerators are proposed to handle the unique problems in GCNs.

HyGCN. HyGCN [49] abstracts GCNs inference as aggregation phases and combination phases and designs two separate acceleration engines forming a hybrid architecture. Aggregation phase shows dynamics and irregularity during executing, HyGCN employs a graph partitioning and window sliding shrinking strategy to reduce unnecessary sparse accesses alleviating the irregular data accesses. The Combination engine leverages traditional systolic array method to exploit various parallelism and highly reusable intervertex data. Finally, HyGCN establishes an interengine pipeline to coordinate these two phases and adopts priority-based memory access to improve overall efficiency.

With respect to the computational characteristics, Auten et al. [184] classifies the inference operations of GCNs into three categories, containing graph traversal, combination, and aggregation. Similar to HyGCN, a modular architecture is proposed to process each operation separately. Specifically, the graph processing element (GPE) is designed to control the graph traversal steps. The DNN accelerator (DNA) is responsible for neural network-liked vertex feature extraction. The aggregator (AGG) buffers the memory requests and accelerates vertex neighbors aggregation.

EnGN. EnGN [42] presents a high-throughput architecture for GCNs, which abstracts GCNs inference as three key phases and accelerates all phases simultaneously. Specifically, EnGN proposes ring edge reduce (REP) dataflow and corresponding RER processing elements to solve the low hardware resource and poor bandwidth utilization. In addition, EnGN adopts an edge reordering strategy to avoid inefficient computation during REP aggregation. Finally, to support large-scale graph processing, EnGN proposes a graph tiling strategy to get subgraphs to enhance locality and fit the on-chip memory.

GRIP. GRIP [185] points out that there are two modes of computation involved in GCN inference, leading to inefficiency and high latency on existing accelerators. To solve this problem, GRIP first decomposes GCN inference into two parts including the execution of edge-centric and vertex-centric and further designs specialized units to accelerate each part. In particular, for the edge-centric stage, GRIP employs parallel prefetching and scaling engines to mitigate memory access irregularities. For the vertex-centric stage, GRIP utilizes a high-speed matrix multiplication model and a specialized memory subsystem for weights to improve reusability. Furthermore, GRIP uses a vertex tiling policy to increase the reuse of weight data to enhance latency.

4.3.3. PIM-Based Graph Learning Accelerators. We have discussed the ASIC and FPGA-based graph learning accelerators above, and in this section, we focus on the emerging PIM architectures. The PIM technology is mainly divided into PNM and PUM. We now discuss them separately.

The idea of PNM is to couple the storage units and the computation units close together physically, which can solve the memory wall problem. In graph learning, the size of the graph data can be enormous, which is far beyond the on-chip memory capacity, requiring substantial memory access. Several PNM accelerators are proposed to solve the memory bottleneck of graph learning.

SmartSAGE. SmartSAGE [186] proposes an in-SSD processing solution to address the memory capacity bottlenecks in the large-scale GCN training. However, blindly transferring in-DRAM processing to the in-SSD processing causes significant performance degradation since the speed gap between the DRAM and SSD. SmartSAGE develops a software/hardware codesign to solve this. In software architecture, it restructures the ML framework to directly access the SSD bypassing the OS page software layers, which significantly reduces the latency to fetch data from the SSD. In hardware architecture, rather than transferring the large, coarse-grained chunks of the input graph from SSD to DRAM for subgraph generation, SmartSAGE offloads the data-intensive phase to the in-storage processing units integrated into the SSD. This allows to only transfer the subgraphs from SSD to DRAM, significantly reducing the data movements.

HolisticGNN. Different from traditional deep learning networks, the GNNs need to handle massive graph data, which is stored in the storage initially and loaded into working memory to reformatted before inference, which brings a significant latency and degrades the performance. HolisticGNN [187] proposes a hardware/software codesign to holistically accelerate GNN inference execution in storage. In software architecture, HolisticGNN manages the data as a graph structure instead of as files directly, which allows to sampling and processing the input data near storage without preprocessing. Besides, HolisticGNN allows programming the tasks using a computational graph and simply transferring it into computational SSD (i.e., CSSD), providing an easy-to-use, programmer-friendly interface. In hardware architecture, HolisticGNN proposes a hardware framework that provides fully programmable FPGA-based fundamental hardware logic for supporting various types of GNN inferences.

On the other hand, PUM involves the computation and storage units together by using some emerging devices (i.e., ReRAM and MRAM) or applying minimal changes to the existing memory architecture (i.e., DRAM and SRAM). Compared to the PNM, PUM has higher computation efficiency due to its in situ processing ability. While the graph learning workloads are dominated by MVM operations, several PUM accelerators are proposed to accelerate it.

PIM-GCN. PIM-GCN [72] is the first in-memory accelerator for GCN and demonstrates the mapping of GCN inference on the ReRAM crossbar architecture. The compute and memory access characteristics of GCN are different from the graph analytics and convolutional neural networks. First of all, the hybrid aggregation and combination execution pattern and workload characteristics of GCN are different from the deep neural network (DNN), which will incur significantly higher memory accesses and computations.

Besides, unlike the conventional graph analytics algorithms, the node feature in GCN is typically much larger than the edge. Thereby, the memory access optimized DNN accelerators and the edge-stationary dataflow graph analytics accelerators are not suitable for the GCN. PIM-GCN proposed a node-stationary dataflow, which maps the node feature data on the ReRAM and leverages the in situ ability of the ReRAM to perform MAC operations for the aggregation and combination for the GCN. To support the compressed sparse formation graph data, PIM-GCN also designs an in-memory traversal mechanism by using the CAM-based search (S-CAM) and CAM-based compare (C-CAM) operations.

TARe. Due to the data characteristics of the graph learning workloads, the scheme of the DNN accelerators that map weights on the ReRAM is not the optimal choice for graph learning. The selection of which data should be mapped on the ReRAM influences the data and writes movement overhead significantly. Unlike the previous work PIM-GCN [72] that fixed the static data on the ReRAM, TARe [75] proposes a task adaptive selection algorithm that selects the static data according to the task and a ReRAM in situ accelerator that supports weight-static, data-static, and hybrid execution mode. For a graph learning workload, the task adaptive selection algorithm first selects the static data and decides the sparse or dense mapping mode. Then the in situ accelerator is configured to work in weight-static, data-static, or hybrid execution mode to achieve higher processing throughput and less data movement.

ReFlip. Unlike the previous works that separate the combination and aggregation stages using different hardware specializations, ReFlip [3] proposes to support both stages of GCN in unified hardware architecture by adopting different mapping schemes. For the combination stage, ReFlip induces a layer-wise mapping scheme that iteratively loads the weights on the crossbar. However, the crossbar utilization may suffer in this way, and ReFlip uses the idle crossbar to exploit the intervertex parallelism by replicating multiple weights copies in PEs, improving both storage and computational efficiency. Due to the incredibly sparse edge data, ReFlip proposes a flipped mapping that maps the vertex feature data on the crossbar and feeds the edge data as input for the aggregation phase. ReFlip also adopts a hybrid row-major and column-major execution model to maximize efficiency and designs locality-aware hardware to minimize energy consumption.

DARe. Different from the others that focus on accelerating the GCN inference phase, DARe [188] focuses on the GCN training phase. In the GCN training phase, DropEdge and Dropout (referred to as DropLayer) operations are implemented to regularize and improve accuracy. However, the DropLayer operation drops different graph and neural units in each iteration, leading to randomly varying traffic patterns that the data exchanged between two adjacent GCN layers keeps changing in each iteration, which is not well-suited for traditional 2D architecture to handle such traffic patterns. DARe proposes a Drop-aware 3D on-chip network integrated ReRAM-based manycore architecture to alleviate the communication bottleneck and improve the computation performance.

5. Challenges and Future Works for Graph Analytics

Domain-Specific High-Level Synthesis. Prototyping the high-performance graph accelerator with hardware description languages (e.g., VHDL and Verilog) is time-consuming and error-prone, and it requires deep knowledge of underlying hardware architecture. To cope with this problem, commercial vendors and academic communities have been actively developing High-level synthesis (HLS) tools, which automatically translate high-level languages into the targeted hardware accelerator with significant design effort reduction. In practice, however, existing general-purpose HLS tools is potentially inefficient for graph applications. It is because that previous HLS tools are domain-agnostic and graph characteristics are not sufficiently considered. As a result, more nontrivial efforts are still needed to fill the programming gap between upper graph applications and underlying efficient hardware accelerators.

Uncertain Patterns for Graph Mining. Most of the graph mining accelerators aim to solve the graph mining problems where the patterns are known as a prior, e.g., subgraph matching, clique finding. The execution plans generated in these accelerators must following the guide of patterns. However, there are still a large number of graph mining applications that cannot provide the patterns as a prior, e.g., frequent subgraph mining. In order to solve these problems, a naïve solution to existing pattern-aware graph mining accelerators is to first enumerate all possible patterns of a certain size and then mine all these patterns. However, this will cause unnecessary workloads because for each pattern the graph data is repeatedly traversed and computed, and there may only exist a small part of the patterns in the graph. Despite that the embedding-centric model used in GRAMER [50] can support this kind of problem, the huge amounts of intermediates will easily consume all the memory capacity of an accelerator. Exploiting the trade-offs of embedding-centric and pattern-aware model on graph mining accelerators may be a possible way to tackle the challenge.

Large Graphs and Patterns for Graph Mining. Compared to real-world scenarios, where the graph usually exceeds billion-scale and the graph pattern can contain hundreds of vertices, existing graph mining accelerator can only handle relatively small workloads. With the size of graphs and patterns increasing, the explosive growth of intermediate data may limit the parallelism of these accelerators because of the limited memory space and extremely load imbalance issue. Some graph mining accelerators that are closely integrated with the CPU can leverage the large main memory to process billion-scale of graphs. However, the data transferring between the accelerator and the host will become a new bottleneck. PIM-based accelerators seem promising facing larger scale graph mining problems. Emerging memory technologies such as 3D memory and nonvolatile memory can provide large capacity and abundant bandwidth and can be extended to construct larger memory systems. However, due to the irregular workload of graph mining, it is nontrivial to achieve good scalability with accelerator

resources. In the future, software/hardware codesign must be considered to fully unleash the power of accelerators to solve real-world problems.

Dynamic Graph Learning. Existing graph learning accelerators are mainly designed for static graphs. However, in many real-world scenarios, graphs are constantly changed. Existing static graph learning accelerators cannot handle such dynamic graphs directly. Therefore, studying efficient graph learning accelerators based on dynamic graph data is an important direction to explore. Compared with static graph learning, dynamic graph learning requires capturing the change of vertices over time, which is supported by introducing RNNs [191–193]. Notably, different types of dynamic graph learning may use different types of RNNs. It means that graph learning accelerators need to support different types of RNN operations in addition to the original graph learning algorithms. This presents a significant challenge to the flexibility and scalability of the accelerator. In addition, although the graph is constantly changing, the graph changing are relative calm [194], i.e., only a few vertices of the graph change in a short period of time. Therefore, recomputing the entire graph would result in a large number of redundant computations. Exploiting the incremental computation for dynamic graph learning accelerators is necessary.

Memory Footprint Limitations. Real-world graph structures usually have a large number of vertices, and the vertices in graph learning usually have hundreds or thousands of dimensions of features. This results in very large datasets for graph learning [195]. Moreover, there are complex dependencies between vertices. These lead to a high computational cost and memory requirements for graph learning. Hence, large-scale distributed accelerator systems for graph learning are well worth investigating. However, due to the irregularity of graphs, task allocation among different accelerators can cause load imbalance and frequent communication overhead. It is crucial to design a reasonable scheduling scheme to minimize the communication overhead and maintain load balancing. In addition, the execution model of the graph learning needs to be carefully designed. Inappropriate execution models can add unnecessary computations. Furthermore, more intermediate data sets are generated, which causes exacerbation of storage requirements. Finally, optimizing memory accesses on the accelerator are also key to further improve system performance. The footprint is larger than the cache size, which causes frequently expensive memory accesses. Exploiting the locality of the graph learning algorithm to improve the cache hit ratio is also worth.

Heterogeneous Graph Learning. A heterogeneous graph refers to the presence of different types of vertex and edges in the graph data [196–198]. In the real world, heterogeneous graphs are more common than homogeneous graphs. Processing heterogeneous graphs are often more complicated than homogeneous graphs. However, existing graph learning accelerators are mainly for homogeneous graph data, and designing accelerators for heterogeneous graph data becomes quite important and urgent. In particular, in a heterogeneous graph, vertices may have different types of features. Most vertices in a heterogeneous graph do not connect all types of other vertices. The sample of each node in the heterogeneous graph requires the selection of strongly correlated neighboring nodes. This leads to more complex sampling than the simple random sampling in homogeneous graphs [196]. The accelerator design is more challenging than the traditional homogeneous graph and requires careful consideration of the execution of complex sampling algorithms.

6. Conclusion

The widespread adoption of graph analytics applications and the gradual increase in the size and complexity of graph data bring significant challenges for software technologies and hardware architectures for graph computing. Several existing software optimization efforts aim to improve the performance and efficiency of graph analytics on general-purpose hardware platforms, such as single-machine platform [84] and distributed platform [97].

However, there is a gap between the characteristics of graph analytics and the hardware features of general-purpose hardware. For example, due to the irregular sparse structure and explosive growth of graph data, these graph applications suffer from inefficient memory systems on traditional architectures. The scale of graph applications continues to grow, and the demand for bandwidth and parallelism has far exceeded what current architectures can provide.

In recent years, novel computing and memory devices have emerged, e.g., FPGAs, HMCs, HBM, and ReRAM, providing massive bandwidth and parallelism resources, making it possible to address bottlenecks in graph applications. In addition, hardware development has been facilitated by open-sourced ISAs, convenient cloud-based hardware development tools, and agile chip development methodologies. These opportunities have inspired a series of research on domain-specific graph accelerators, pursuing extreme performance and power efficiency on different architectures.

Software optimization technologies and hardware acceleration technologies have achieved significant performance improvements. However, the majority of graph computing in real-world scenarios is characterized by dynamic changes and complex and diverse application requirements (such as graph queries, graph processing, subgraph matching, graph neural network training, and inference) [2, 94, 137, 152]. This brings new requirements and challenges to graph computing in terms of basic theory, key technologies of system software, and architecture.

This paper systematically introduces and discusses the research progress and trends of key technologies of software systems implementation and domain-specific architectures for graph analytics, including graph processing, graph mining, and graph learning. This paper presents the current research status and compares the research progress. Finally, we also point out the challenges faced by graph analytics. In conclusion, graph analytics is still a popular research topic with many challenges and opportunities. We hope that this paper will help more researchers and engineers to

understand and participate in the research of software implementation and domain-specific architecture techniques for graph analytics systems and to collaborate to address these challenges.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this article.

Acknowledgments

This paper is supported by National Natural Science Foundation of China under grant No. 61832006, 61825202, 62072193, and Major Scientific Project of Zhejiang Lab No. 2022PI0AC03. Yu Zhang (zhyu@hust.edu.cn) is the corresponding author of this paper.

References

[1] N. Liu, D. S. Li, Y. M. Zhang, and X. L. Li, "Large-scale graph processing systems: a survey," *Frontiers of Information Technology & Electronic Engineering*, vol. 21, no. 3, pp. 384–404, 2020.

[2] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming and dynamic graphs: concepts, models, systems, and parallelism," 2020, https://arxiv.org/abs/1912.12740.

[3] Y. Huang, L. Zheng, P. Yao et al., "Accelerating graph convolutional networks using crossbar-based processing in-memory architectures," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1–14, Seoul, Korea, Republic of, 2022.

[4] O. Zorzi, "Granovetter (1983): The strength of weak ties: a network theory revisited," in *Schlu¨sselwerke der Netzwerkforschung*, pp. 243–246, Springer, 2019.

[5] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[6] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp, "Biomolecular network motif counting and discovery by color coding," *Bioinformatics*, vol. 24, no. 13, pp. i241–i249, 2008.

[7] Y.-R. Cho and A. Zhang, "Predicting protein function by frequent functional association pattern mining in protein interaction networks," *IEEE Transactions on Information Technology in Biomedicine*, vol. 14, no. 1, pp. 30–36, 2010.

[8] B. Gaüzere, L. Brun, and D. Villemin, "Two new graphs kernels in chemoinformatics," *Pattern Recognition Letters*, vol. 33, no. 15, pp. 2038–2047, 2012.

[9] H. Kashima, H. Saigo, M. Hattori, and K. Tsuda, "Graph Kernels for Chemoinformatics," in *Chemoinformatics and advanced machine learning perspectives: complex computational methods and collaborative techniques*, pp. 1–15, IGI global, 2011.

[10] P. Grnarova, K. Y. Levy, A. Lucchi et al., "A domain agnostic measure for monitoring and evaluating gans," in *Proceedings of the 2019 Thirty-third Conference on Neural Information Processing*, pp. 12 069–12 079, Vancouver, BC, Canada, 2019.

[11] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2021.

[12] A. Sperduti and A. Starita, "Supervised neural networks for the classification of structures," *IEEE Transactions on Neural Networks*, vol. 8, no. 3, pp. 714–735, 1997.

[13] R. Waleffe, J. Mohoney, T. Rekatsinas, and S. Venkataraman, "Marius++: large-scale training of graph neural networks on a single machine," 2022, https://arxiv.org/abs/2202.02365.

[14] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur, "Protein interface prediction using graph convolutional networks," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[15] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 974–983, London, United Kingdom, 2018.

[16] H. Dai, Z. Kozareva, B. Dai, A. Smola, and L. Song, "Learning steady-states of iterative algorithms over graphs," in *Proceedings of the Thirty-fifth International Conference on Machine Learning*, pp. 1106–1114, Stockholm, Sweden, 2018.

[17] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He, "An efficient graph accelerator with parallel data conflict management," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018*, pp. 1–12, Limassol, Cyprus, November 01-04, 2018.

[18] L. Bindschaedler, J. Malicevic, B. Lepers, A. Goel, and W. Zwaenepoel, "Tesseract: distributed, general graph pattern mining on evolving graphs," in *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 458–473, Online Event, United Kingdom, 2021.

[19] A. M. Caulfield, E. S. Chung, A. Putnam et al., "Configurable clouds," *IEEE Micro*, vol. 37, no. 3, pp. 52–61, 2017.

[20] Amazon, "Amazon F1 cloud," 2020. [Online]. Available: https://aws.amazon.com/cn/ec2/instance-types/f1/.

[21] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: exploring large scale graph processing on multi-FPGA architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*, pp. 217–226, Monterey, California, USA, 2017.

[22] G. Dai, Y. Chi, Y. Wang, and H. Yang, "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 (FPGA'16)*, pp. 105–110, Monterey, California, USA, 2016.

[23] Z. Shao, R. Li, D. Hu, X. Liao, and H. Jin, "Improving performance of graph processing on FPGA-DRAM platform by two-level vertex caching," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)*, pp. 320–329, Seaside, CA, USA, 2019.

[24] Q. Wang, L. Zheng, J. Zhao, X. Liao, H. Jin, and J. Xue, "A conflict-free scheduler for High-performance graph processing on multi-pipeline FPGAs," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 2, pp. 1–14: 26, 2020.

[25] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "HitGraph: high-throughput graph processing framework on FPGA," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2249–2264, 2019.

[26] E. Nurvitadhi, G. Weisz, Y. Wang et al., "Graphgen: an FPGA framework for vertex-centric graph computation," in *Proceedings of the 22nd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'14)*, pp. 25–28, Boston, MA, USA, 2014.

[27] T. Oguntebi and K. Olukotun, "Graphops: a dataflow library for graph analytics acceleration," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'16)*, pp. 111–117, Monterey, California, USA, 2016.

[28] M. Besta, M. Fischer, T. Ben-Nun, J. de Fine Licht, and T. Hoefler, "Substream-centric maximum matchings on FPGA," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)*, pp. 152–161, Seaside, CA, USA, 2019.

[29] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Proceedings of the 2011 IEEE Hot Chips 23 Symposium (HCS'11)*, pp. 1–24, Stanford, CA, USA, 2011.

[30] J. Kim and Y. Kim, "HBM: memory solution for bandwidth-hungry processors," in *Proceedings of the 2014 IEEE Hot Chips 26 Symposium (HCS'14)*, pp. 1–24, Cupertino, CA, 2014.

[31] J. Zhang, S. Khoram, and J. Li, "Boosting the performance of FPGA-based graph processor using hybrid memory cube: a case for breadth first search," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*, pp. 207–216, Monterey, California, USA, 2017.

[32] J. Zhang and J. Li, "Degree-aware hybrid graph traversal on FPGA-HMC platform," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*, pp. 229–238, Monterey, CALIFORNIA, USA, 2018.

[33] S. Khoram, J. Zhang, M. Strange, and J. Li, "Accelerating graph analytics by co-optimizing storage and access on an FPGA-HMC platform," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*, pp. 239–248, Monterey, CALIFORNIA, USA, 2018.

[34] C. Liu, Z. Shao, K. Li et al., "ScalaBFS: a scalable BFS accelerator on FPGA-HBM platform," in *Proceedings of the 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'21),*, p. 147, Virtual Event, USA, 2021.

[35] Y. Umuroglu, D. Morrison, and M. Jahre, "Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform," in *Proceedings of the 25th International Conference on Field Programmable Logic and Applications (FPL'15)*, pp. 1–8, London, UK, 2015.

[36] L. Remis, M. J. Garzaran, R. Asenjo, and A. Navarro, "Breadth-first search on heterogeneous platforms: a case of study on social networks," in *Proceedings of the 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'16)*, pp. 118–125, Los Angeles, CA, USA, 2016.

[37] H. Zeng and V. K. Prasanna, "Graphact: accelerating GCN training on CPU-FPGA heterogeneous platforms," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*, pp. 255–265, Seaside, CA, USA, 2020.

[38] Y. Zou and M. Lin, "Gridgas: an I/O-efficient heterogeneous FPGA+CPU computing platform for very large-scale graph analytics," in *Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT'18)*, pp. 246–249, Naha, Japan, 2018.

[39] S. Zhou and V. K. Prasanna, "Accelerating graph analytics on CPU-FPGA heterogeneous platform," in *Proceedings of the 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'17)*, pp. 137–144, Campinas, Brazil, 2017.

[40] O. G. Attia, T. Johnson, K. Townsend, P. H. Jones, and J. Zambreno, "Cygraph: a reconfigurable architecture for parallel breadth-first search," in *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW'14)*, pp. 228–235, Phoenix, AZ, USA, 2014.

[41] O. Kalinsky, B. Kimelfeld, and Y. Etsion, "The triejax architecture: accelerating graph operations through relational joins," in *Proceedings of the 2020 Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, pp. 1217–1231, Lausanne, Switzerland, 2020.

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

[43] Q. Chen, B. Tian, and M. Gao, "FINGERS: exploiting fine-grained parallelism in graph mining accelerators," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS'22)*, pp. 43–55, Lausanne, Switzerland, 2022.

[44] Y. Zhang, X. Liao, H. Jin et al., "Depgraph: a dependency-driven accelerator for efficient iterative graph processing," in *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*, pp. 371–384, Seoul, Korea (South), 2021.

[45] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, pp. 578–592, Williamsburg, VA, USA, 2018.

[46] A. Basak, S. Li, X. Hu et al., "Analysis and optimization of the memory hierarchy for graph processing workloads," in *Proceedings of the 25th IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*, pp. 373–386, Washington, DC, USA, 2019.

[47] M. Yan, X. Hu, S. Li et al., "Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture(MICRO'19)*, pp. 615–628, Columbus, OH, USA, 2019.

[48] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: a highperformance and energy-efficient accelerator for graph analytics," in *2016 49th annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pp. 1–13, Taipei, Taiwan, 2016.

[49] M. Yan, L. Deng, X. Hu et al., "Hygcn: a GCN accelerator with hybrid architecture," in *Proceedings of the 2020 IEEE International Symposium on High Performance Computer*

Architecture (HPCA'20), pp. 15–29, San Diego, CA, USA, 2020.

[50] P. Yao, L. Zheng, Z. Zeng et al., "A locality-aware energy-efficient accelerator for graph mining applications," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 895–907, Athens, Greece, 2020.

[51] M. M. Ozdal, S. Yesil, T. Kim et al., "Energy efficient architecture for graph analytics accelerators," in *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*, pp. 166–177, Seoul, Republic of Korea, 2016.

[52] V. Dadu, S. Liu, and T. Nowatzki, "Polygraph: exposing the value of flexibility for graph processing accelerators," in *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'21)*, pp. 595–608, Valencia, Spain, 2021.

[53] J. Li, A. Louri, A. Karanth, and R. C. Bunescu, "GCNAX: a flexible and energy-efficient accelerator for graph convolutional neural networks," in *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*, pp. 775–788, Seoul, Korea (South), 2021.

[54] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and A. Arvind, "Flexminer: a pattern–aware accelerator for graph pattern mining," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 581–594, Valencia, Spain, 2021.

[55] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*, pp. 234–248, San Diego, CA, USA, 2020.

[56] P. Chi, S. Li, C. Xu et al., "PRIME: a novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, vol. 44no. 3, pp. 27–39, Seoul, Republic of Korea, 2016.

[57] A. Shafiee, A. Nag, N. Muralimanohar et al., "ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 14–26, Seoul, Republic of Korea, 2016.

[58] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: a pipelined reram-based accelerator for deep learning," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 541–552, Austin, TX, USA, 2017.

[59] M. Wilkening, U. Gupta, S. Hsia et al., "Recssd: Near data processing for solid state drive based recommendation inference," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 717–729, Virtual, USA, 2021.

[60] Y. Kwon, Y. Lee, and M. Rhu, "Tensordimm: a practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 740–753, Columbus, OH, USA, 2019.

[61] L. Ke, U. Gupta, B. Y. Cho et al., "Recnmp: accelerating personalized recommendation with near-memory processing," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 790–803, Valencia, Spain, 2020.

[62] J. Chen, G. Lin, J. Chen, and Y. Wang, "Towards efficient allocation of graph convolutional networks on hybrid computation-in-memory architecture," *Science China Information Sciences*, vol. 64, no. 6, pp. 1–14, 2021.

[63] X. Qian, "Graph processing and machine learning architectures with emerging memory technologies: a survey," *Science China Information Sciences*, vol. 64, no. 6, pp. 1–25, 2021.

[64] V. Seshadri, D. Lee, T. Mullins et al., "Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 273–287, Boston, MA, USA, 2017.

[65] X. Xin, Y. Zhang, and J. Yang, "ELP2IM: efficient and low power bitwise operation processing in DRAM," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 303–314, San Diego, CA, USA, 2020.

[66] N. Hajinazar, G. F. Oliveira, S. Gregorio et al., "SIMDRAM: a framework for bit-serial SIMD processing using DRAM," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 329–345, Virtual, USA, 2021.

[67] C. Xu, D. Niu, Y. Zheng, S. Yu, and Y. Xie, "Impact of cell failure on reliable cross-point resistive memory design," *ACM Transactions on Design Automation of Electronic Systems*, vol. 20, no. 4, pp. 1–63: 21, 2015.

[68] G. Yuan, P. Behnam, Z. Li et al., "FORMS: fine-grained polarized reram-based in-situ computation for mixed–signal DNN accelerator," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 265–278, Valencia, Spain, 2021.

[69] L. Song, Y. Zhuo, X. Qian, H. H. Li, and Y. Chen, "Graphr: accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 531–543, Vienna, Austria, 2018.

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

[71] M. Besta, R. Kanakagiri, G. Kwasniewski et al., "SISA: set-centric instruction set architecture for graph mining on processing-in-memory systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 282–297, Virtual Event, Greece, 2021.

[72] N. Challapalle, K. Swaminathan, N. Chandramoorthy, and V. Narayanan, "Crossbar based processing in memory accelerator architecture for graph convolutional networks," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, Munich, Germany, 2021.

[73] S. Angizi and D. Fan, "Graphide: a graph processing accelerator leveraging in-dram-computing," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pp. 45–50, Tysons Corner, VA, USA, 2019.

[74] L. Zheng, J. Zhao, Y. Huang et al., "Spara: an energy-efficient reram-based accelerator for sparse graph analytics applications," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 696–707, New Orleans, LA, USA, 2020.

[75] Y. He, Y. Wang, C. Liu, H. Li, and X. Li, "Tare: task-adaptive in-situ reram computing for graph learning," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 577–582, San Francisco, CA, USA, 2021.

[76] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3d-stacked DRAM," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 131–143, Portland, Oregon, 2015.

[77] V. T. Lee, A. Mazumdar, C. C. del Mundo, A. Alaghi, L. Ceze, and M. Oskin, "POSTER: application-driven near-data processing for similarity search," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 132-133, Portland, OR, USA, 2017.

[78] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 105–117, Portland, OR, USA, 2015.

[79] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: enabling instruction-level PIM offloading in graph computing frameworks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 457–468, Austin, TX, USA, 2017.

[80] L. He, C. Liu, Y. Wang, S. Liang, H. Li, and X. Li, "Gcim: a near-data processing accelerator for graph construction," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 205–210, San Francisco, CA, USA, 2021.

[81] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 113–124, San Francisco, CA, USA, 2015.

[82] M. Zhang, Y. Zhuo, C. Wang et al., "Graphp: reducing communication for pim-based graph processing with efficient data partition," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 544–557, Vienna, Austria, 2018.

[83] Y. Zhuo, C. Wang, M. Zhang et al., "Graphq: scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 712–725, Columbus, OH, USA, 2019.

[84] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 135–146, Shenzhen, China, 2013.

[85] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pp. 456–471, Farminton, Pennsylvania, 2013.

[86] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming*, pp. 183–193, San Francisco, CA, USA, 2015.

[87] Y. Zhang, X. Liao, H. Jin, L. Gu, G. Tan, and B. B. Zhou, "HotGraph: efficient asynchronous processing for real-world graphs," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 799–809, 2017.

[88] Y. Zhang, X. Liao, H. Jin et al., "Cgraph: a correlations-aware approach for efficient concurrent iterative graph processing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18),*, pp. 441–452, Boston, MA, 2018.

[89] Y. Zhang, J. Zhao, X. Liao et al., "Cgraph: a distributed storage and processing system for concurrent iterative graph analysis jobs," *ACM Transactions on Storage*, vol. 15, no. 2, pp. 10: 1–10: 26, 2019.

[90] M. Mariappan and K. Vora, "Graphbolt: dependency-driven synchronous processing of streaming graphs," in *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–16, Dresden, Germany, 2019.

[91] M. Mariappan, J. Che, and K. Vora, "Dzig: sparsity-aware incremental processing of streaming graphs," in *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 83–98, Online Event, United Kingdom, 2021.

[92] X. Jiang, C. Xu, X. Yin, Z. Zhao, and R. Gupta, "Tripoline: generalized incremental graph processing via graph triangle inequality," in *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 17–32, Online Event, United Kingdom, 2021.

[93] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: a high-performance graph processing library on the GPU," in *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, pp. 1–12, Barcelona, Spain, 2016.

[94] Y. Zhang, X. Liao, H. Jin, B. He, H. Liu, and L. Gu, "Digraph: an efficient path-based iterative directed graph processing system on multiple GPUs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 601–614, Providence, RI, USA, 2019.

[95] L. Zheng, X. Li, Y. Zheng et al., "Scaph: scalable {gpu-accelerated} graph processing with {value-driven}- differential scheduling," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 573–588, Portland, OR, USA, 2020.

[96] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: large-scale graph computation on just a PC," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pp. 31–46, Hollywood, CA, USA, 2012.

[97] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 472–488, Farminton, Pennsylvania, 2013.

[98] X. Zhu, W. Han, and W. Chen, "Gridgraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 375–386, Santa Clara, CA, 2015.

[99] P. Yuan, C. Xie, L. Liu, and H. Jin, "PathGraph: a path centric graph processing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2998–3012, 2016.

[100] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: processing a trillion-edge graph on a single machine," in *Proceedings of the Twelfth European Conference on Computer Systems*, pp. 527–543, Belgrade, Serbia, 2017.

[101] J. Zhao, Y. Zhang, X. Liao et al., "Graphm: an efficient storage system for high throughput of concurrent graph processing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 3: 1–3: 14, Denver, Colorado, 2019.

[102] Y. Zhang, Y. Liang, J. Zhao et al., "Egraph: efficient concurrent GPU-based dynamic graph processing," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–14, 2022.

[103] G. Malewicz, M. H. Austern, A. J. Bik et al., "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, Indianapolis, Indiana, USA, 2010.

[104] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[105] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph–parallel computation on natural graphs," in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pp. 17–30, Hollywood, CA, 2012.

[106] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "SYNC or async: time to fuse for distributed graph-parallel computation," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 194–204, 2015.

[107] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 8, pp. 2091–2100, 2014.

[108] K. Vora, R. Gupta, and G. Xu, "Kickstarter: fast and accurate computations on streaming graphs via trimmed approximations," in *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*, pp. 237–251, Xi'an, China, 2017.

[109] S. Gong, C. Tian, Q. Yin et al., "Automating incremental graph processing with flexible memoization," *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1613–1625, 2021.

[110] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: scale-out graph processing from secondary storage," in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 410–424, Monterey, California, 2015.

[111] Y. Zhang, X. Liao, L. Gu et al., "Asyngraph: maximizing data parallelism for efficient iterative graph processing on GPUs," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 4, pp. 29: 1–29: 21, 2020.

[112] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: minimizing data transfer during out-of-GPU-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, Heraklion, Greece, 2020.

[113] K. Vora, "Lumos: dependency-driven disk-based graph processing," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 429–442, Renton, WA, 2019.

[114] Y. Zhang, X. Liao, X. Shi, H. Jin, and B. He, "Efficient disk-based directed graph processing: a strongly connected component approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 830–842, 2018.

[115] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland: A novel abstraction-based out-of-core graph processing system," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 608–621, 2018.

[116] P. Pan and C. Li, "Congra: towards efficient processing of concurrent graph queries on shared–memory machines," in *Proceedings of the 2017 IEEE International Conference on Computer Design*, pp. 217–224, Boston, MA, USA, 2017.

[117] P. Pan, C. Li, and M. Guo, "CongraPlus: towards efficient processing of concurrent graph queries on NUMA machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 1990–2002, 2019.

[118] X. Liao, J. Zhao, Y. Zhang et al., "A structure-aware storage optimization for out-of-core concurrent graph processing," *IEEE Transactions on Computers*, vol. 71, no. 7, pp. 1612–1625, 2022.

[119] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: a resilient distributed graph system on spark," in *First international workshop on graph data management experiences and systems*, pp. 1–6, New York, New York, 2013.

[120] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 3, pp. 1–39, 2019.

[121] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: a computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 301–316, Savannah, GA, 2016.

[122] W. Fan, J. Xu, Y. Wu, W. Yu, and J. Jiang, "GRAPE: Parallelizing sequential graph computations," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1889–1892, 2017.

[123] Y. Zhang, X. Liao, H. Jin, L. Gu, and B. B. Zhou, "FBSgraph: accelerating asynchronous graph processing via forward and backward sweeping," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 5, pp. 895–907, 2018.

[124] Q. Wang, Y. Zhang, H. Wang et al., "Automating incremental and asynchronous evaluation for recursive aggregate data processing," in *Proceedings of the 2020 International Conference on Management of Data*, pp. 2439–2454, Portland, OR, USA, 2020.

[125] R. Cheng, J. Hong, A. Kyrola et al., "Kineograph: taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 85–98, Bern, Switzerland, 2012.

[126] X. Shi, B. Cui, Y. Shao, and Y. Tong, "Tornado: a system for real-time iterative analysis over evolving data," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 417–430, San Francisco, California, USA, 2016.

[127] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 425–440, Monterey, California, 2015.

[128] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour, "Scalemine: scalable parallel frequent subgraph mining in a single large graph," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 716–727, Salt Lake City, UT, USA, 2016.

[129] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng, "G-miner: an efficient task-oriented graph mining system," in *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–12, Porto, Portugal, 2018.

[130] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, "Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 763–782, Carlsbad, CA, 2018.

[131] V. Dias, C. H. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy, "Fractal: a general-purpose graph pattern mining system," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 1357–1374, Amsterdam, Netherlands, 2019.

[132] D. Yan, G. Guo, M. M. Chowdhury, M. T. Özsu, W. S. Ku, and J. C. Lui, "G-thinker: a distributed framework for mining subgraphs in a big graph," in *2020 IEEE 36th International*

Conference on Data Engineering (ICDE), pp. 1369–1380, Dallas, TX, USA, 2020.

[133] X. Chen, R. Dathathri, G. Gill, and K. Pingali, "Pangolin: An efficient and flexible graph mining system on cpu and gpu," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1190–1205, 2020.

[134] V. Trigonakis, J. P. Lozi, T. Faltín et al., "ADFS: an almost depth-first-search distributed graph-querying system," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 209–224, Portland, OR, USA, 2021.

[135] D. Mawhirter and B. Wu, "Automine: harmonizing high-level abstraction and high performance for graph mining," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 509–523, Huntsville, Ontario, Canada, 2019.

[136] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu, "Graphzero: A high-performance subgraph matching system," *ACM SIGOPS Operating Systems Review*, vol. 55, no. 1, pp. 21–37, 2021.

[137] K. Jamshidi, R. Mahadasa, and K. Vora, "Peregrine: a pattern-aware graph mining system," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, Heraklion, Greece, 2020.

[138] T. Shi, M. Zhai, Y. Xu, and J. Zhai, "Graphpi: high performance graph pattern matching through effective redundancy elimination," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, Atlanta, GA, USA, 2020.

[139] J. Chen and X. Qian, "Dwarvesgraph: a high-performance graph mining system with pattern decomposition," 2020, https://arxiv.org/abs/2008.09682.

[140] J. Chen and X. Qian, "Kudu: an efficient and scalable distributed graph pattern mining engine," 2021, https://arxiv.org/abs/2105.03789.

[141] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali, "Sandslash: a two-level framework for efficient graph pattern mining," in *Proceedings of the ACM International Conference on Supercomputing*, pp. 378–391, Virtual Event, USA, 2021.

[142] C. Gui, X. Liao, L. Zheng, P. Yao, Q. Wang, and H. Jin, "Sumpa: efficient pattern-centric graph mining with pattern abstraction," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 318–330, Atlanta, GA, USA, 2021.

[143] X. Chen, J. Bielak, Q. Ning et al., "Efficient and scalable graph pattern mining on GPUs," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 857–877, Carlsbad, CA, 2022.

[144] M. Y. Wang, "Deep graph library: towards efficient and scalable deep learning on graphs," in *Proceedings of the 2019 International Conference on Learning Representations*, pp. 1–14, New Orleans, LA, USA, 2019.

[145] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," pp. 1–17, 2019, https://arxiv.org/abs/1903.02428.

[146] Y. Hu, Z. Ye, M. Wang et al., "Featgraph: a flexible and efficient backend for graph neural network systems," in *Proceedings of the 2020 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, Atlanta, GA, USA, 2020.

[147] H. Liu, S. Lu, X. Chen, and B. He, "G^3 when graph neural networks meet parallel graph processing systems on GPUs," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2813–2816, 2020.

[148] Y. Wang, B. Feng, G. Li et al., "GNNadvisor: an adaptive and efficient runtime system for {GNN} acceleration on {gpus}," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 515–531, Carlsbad, CA, USA, 2021.

[149] L. Ma, Z. Yang, Y. Miao et al., "Neugraph: parallel deep neural network computation on large graphs," in *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 443–458, Renton, WA, 2019.

[150] S. Min, K. Wu, S. Huang et al., "Large graph convolutional network training with GPU-oriented data communication architecture," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2087–2100, 2021.

[151] J. Yang, D. Tang, X. Song et al., "GNNlab: a factored system for sample-based GNN training over GPUs," in *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 417–434, Rennes, France, 2022.

[152] S. Gandhi and A. P. Iyer, "P^3: distributed deep graph learning at scale," in *Proceedsings of the 15th USENIX Symposium on Operating Systems Design and Implementation*, pp. 551–568, Carlsbad, CA, USA, 2021.

[153] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with roc," in *Proceedings of the 2020 Machine Learning and Systems*, pp. 1–12, Carlsbad, CA, USA, 2020.

[154] V. Md, S. Misra, G. Ma et al., "DistGNN: scalable distributed training for large-scale graph neural networks," in *Proceedings of the 2021 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 76: 1–76: 14, St. Louis, Missouri, 2021.

[155] C. Zheng, H. Chen, Y. Cheng et al., "ByteGNN: efficient graph neural network training at large scale," in *Proceedings of the VLDB Endowment*, pp. 1228–1242, Carlsbad, CA, USA, 2022.

[156] Q. Wang, Y. Zhang, H. Wang, C. Chen, X. Zhang, and G. Yu, "Neutronstar: distributed GNN training with hybrid dependency management," in *Proceedings of the 2022 International Conference on Management of Data*, pp. 1301–1315, Philadelphia, PA, USA, 2022.

[157] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Message passing neural networks," in *Machine learning meets quantum physics*, pp. 199–214, Springer, 2020.

[158] R. Zhu, K. Zhao, H. Yang et al., "Aligraph: a comprehensive graph neural network platform," 2019, https://arxiv.org/abs/1902.08730.

[159] N. A. Khan, O. I. Khalaf, C. A. T. Romero, M. Sulaiman, and M. A. Bakar, "Application of Euler neural networks with soft computing paradigm to solve nonlinear problems arising in heat transfer," *Entropy*, vol. 23, no. 8, p. 1053, 2021.

[160] J. Thorpe, Y. Qiao, J. Eyolfson et al., "Dorylus: affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*, pp. 495–514, Carlsbad, CA, USA, 2021.

[161] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, "DGCL: an efficient communication library for distributed GNN training," in *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 130–144, Online Event, United Kingdom, 2021.

[162] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: exploring large-scale graph processing on multi-FPGA architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017*, pp. 217–226, Monterey, CA, USA, February 22-24, 2017.

[163] X. Chen, H. Tan, Y. Chen, B. He, W. Wong, and D. Chen, "Thundergp: Hls-based graph processing framework on FPGAs," in *FPGA '21: The 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 69–80, Virtual Event, USA, February 28 - March 2, 2021.

[164] Y. Chi, L. Guo, and J. Cong, "Accelerating SSSP for power-law graphs," in *FPGA '22: The 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 190–200, Virtual Event, USA, 2022.

[165] N. Sundaram, N. R. Satish, M. M. Patwary et al., "Graphmat: high performance graph analytics made productive," 2015, https://arxiv.org/abs/1503.07241.

[166] Z. Dan, M. Xiaoyu, T. Michael, and C. Derek, "Minnow: lightweight offload engines for worklist management and worklist-directed prefetching," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS'18)*, pp. 593–607, Williamsburg, VA, USA, 2018.

[167] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*, pp. 1–14, Fukuoka, Japan, 2018.

[168] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 178–190, Waikiki, Hawaii, 2015.

[169] S. Rahman, N. B. Abu-Ghazaleh, and R. Gupta, "Graphpulse: an event-driven hardware accelerator for asynchronous graph processing," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*, pp. 908–921, Athens, Greece, 2020.

[170] A. Mukkara, N. Beckmann, and D. Sanchez, "Phi: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1009–1022, Columbus, OH, USA, 2019.

[171] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: Understanding graph computing in the context of industrial solutions," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, Austin, TX, USA, 2015.

[172] N. Challapalle, S. Rampalli, L. Song et al., "Gaas-x: Graph analytics accelerator supporting sparse data representation using crossbar architectures," in *2020 ACM/IEEE 47th annual international symposium on computer architecture (ISCA'20)*, pp. 433–445, Valencia, Spain, 2020.

[173] Q. Wang, L. Zheng, Y. Huang et al., "Grasu: a fast graph update library for FPGA-based dynamic graph processing," in *FPGA '21: The 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 149–159, Virtual Event, USA, February 28 - March 2, 2021.

[174] A. Andrey, Y. Serif, O. M. Mustafa, K. Taemin, B. Steven, and O. Ozcan, "A template-based design methodology for graph-parallel hardware accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems(TCAD)*, vol. 37, no. 2, pp. 420–430, 2018.

[175] Y. Huang, L. Zheng, P. Yao et al., "A heterogeneous PIM hardware-software co-design for energy-efficient graph processing," in *2020 IEEE international parallel and distributed processing symposium (IPDPS'20)*, pp. 684–695, New Orleans, LA, USA, 2020.

[176] X. Jin, Z. Yang, X. Lin, S. Yang, L. Qin, and Y. Peng, "Fast: FPGA-based subgraph matching on massive graphs," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 1452–1463, Chania, Greece, 2021.

[177] G. Rao, J. Chen, J. Yik, and X. Qian, "Sparsecore: stream isa and processor specialization for sparse computation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 186–199, Lausanne, Switzerland, 2022.

[178] N. Talati, H. Ye, Y. Yang et al., "Ndminer: accelerating graph pattern mining using near data processing," in *Proceedings of the 49th ACM/IEEE Annual International Symposium on Computer Architecture, ser. ISCA 2022*, pp. 1–14, New York, NY, USA, 2022.

[179] G. Dai, Z. Zhu, T. Fu et al., "dimmining: pruning-efficient and parallel graph mining on dimm-based near-memory-computing," in *Proceedings of the 49th ACM/IEEE Annual International Symposium on Computer Architecture, ser. ISCA 2022*, pp. 1–14, New York, NY, USA, 2022.

[180] G. Rao, J. Chen, and X. Qian, "Intersectx: an efficient accelerator for graph mining," 2020, https://arxiv.org/abs/2012.10848.

[181] T. Geng, A. Li, R. Shi et al., "AWB-GCN: a graph convolutional network accelerator with runtime workload rebalancing," in *53rd Annual IEEE/ACM International Symposium on Microarchitec-Ture, MICRO 2020*, pp. 922–936, Athens, Greece, 2020.

[182] T. Geng, C. Wu, Y. Zhang et al., "I-GCN: a graph convolutional network accelerator with runtime locality enhancement through islandization," in *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event*, pp. 1051–1063, Virtual Event, Greece, 2021.

[183] J. Romero Hung, C. Li, P. Wang et al., "Ace-gcn: A fast data-driven FPGA accelerator for gcn embedding," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 14, no. 4, 2021.

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

[185] K. Kiningham, P. Levis, and C. Ré, "GRIP: a graph neural network accelerator architecture," 2020, https://arxiv.org/abs/2007.13828.

[186] Y. Lee, J. Chung, and M. Rhu, "Smartsage: Training large-scale graph neural networks using in-storage processing architectures," in *Proceedings of the 49th IEEE/ACM International Symposium on Computer Architecture (ISCA-49)*, pp. 1–14, New York, New York, 2022.

[187] M. Kwon, D. Gouk, S. Lee, and M. Jung, "Hardware/software co-programmable framework for computational ssds to accelerate deep learning service on large-scale graphs," in *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST' 22)*, pp. 1–14, Santa Clara, CA, 2022.

[188] A. I. Arka, B. K. Joardar, J. R. Doppa, P. P. Pande, and K. Chakrabarty, "Dare: droplayer-aware manycore reram

architecture for training graph neural networks," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, Munich, Germany, 2021.

[189] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. K. Prasanna, "Accurate, efficient and scalable graph embedding," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 462–471, Rio de Janeiro, Brazil, 2019.

[190] J. R. Hung, C. Li, T. Wang et al., "Dragon: dynamic recurrent accelerator for graph online convolution," in *ACM Transactions on Design Automation of Electronic Systems (TODAES*, pp. 1084–4309, Portland, OR, USA, 2022.

[191] J. Wu, R. Zhang, Y. Mao, H. Guo, M. Soflaei, and J. Huai, "Dynamic graph convolutional networks for entity linking," in *Proceeding of the 2020 World Wide Web Conference, WWW'20*, pp. 1149–1159, Taipei, Taiwan, 2020.

[192] A. Pareja, G. Domeniconi, J. Chen et al., "Evolvegcn: Evolving graph convolutional networks for dynamic graphs," in *Proceeding of the 2020 Conference on Artificial Intelligence (AAAI'20)*, pp. 5363–5370, Portland, OR, USA, 2020.

[193] O. A. Malik, S. Ubaru, L. Horesh, M. E. Kilmer, and H. Avron, "Dynamic graph convolutional networks using the tensor m-product," in *Proceedings of the 2021 SIAM International Conference on Data Mining, SDM'21*, pp. 729–737, Portland, OR, USA, 2021.

[194] K. Gabert, K. Sancak, M. Y. Özkaya, A. Pinar, and Ü. V. Çatalyürek, "EIGA: elastic and scalable dynamic graph analysis," in *Proceedings of the 2021 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'21)*, pp. 50: 1–50: 15, St. Louis, Missouri, 2021.

[195] W. Hu, M. Fey, H. Ren, M. Nakata, Y. Dong, and J. Leskovec, "OGB-LSC: a large-scale challenge for machine learning on graphs," in *Proceedings of the 2021 Neural Information Processing Systems Track on Datasets and Benchmarks*, Portland, OR, USA, 2021.

[196] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'19)*, pp. 793–803, Anchorage, AK, USA, 2019.

[197] X. Wang, H. Ji, C. Shi et al., "Heterogeneous graph attention network," in *Proceeding of the 2019 World Wide Web Conference, WWW'19*, pp. 2022–2032, San Francisco, CA, USA, 2019.

[198] T. Yang, L. Hu, C. Shi, H. Ji, X. Li, and L. Nie, "HGAT: heterogeneous graph attention networks for semi-supervised short text classification," *ACM Transactions on Information Systems*, vol. 39, no. 3, pp. 32: 1–32: 29, 2021.