Gui CY, Zheng L, He BS *et al.* A survey on graph processing accelerators: Challenges and opportunities. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 34(2): 339–371 Mar. 2019. DOI 10.1007/s11390-019-1914-z

A Survey on Graph Processing Accelerators: Challenges and Opportunities

Chuang-Yi Gui[1,2,3], *Student Member, CCF*, Long Zheng[1,2,3,*], *Member, CCF, ACM, IEEE*
Bingsheng He[4], *Senior Member, IEEE, Member, ACM*, Cheng Liu[4,5], Xin-Yu Chen[4]
Xiao-Fei Liao[1,2,3], *Senior Member, CCF, Member, IEEE*, and Hai Jin[1,2,3], *Fellow, CCF, IEEE, Member, ACM*

[1] *National Engineering Research Center for Big Data Technology and System, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*

[2] *Services Computing Technology and System Laboratory, School of Computer Science and Technology Huazhong University of Science and Technology, Wuhan 430074, China*

[3] *Cluster and Grid Computing Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*

[4] *School of Computing, National University of Singapore, Singapore 117418, Singapore*

[5] *Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*

E-mail: {chygui, longzh}@hust.edu.cn; hebs@comp.nus.edu.sg; liucheng@ict.ac.cn; xinyuc@comp.nus.edu.sg
{xfliao, hjin}@hust.edu.cn

Received July 16, 2018; revised February 2, 2019.

Abstract Graph is a well known data structure to represent the associated relationships in a variety of applications, e.g., data science and machine learning. Despite a wealth of existing efforts on developing graph processing systems for improving the performance and/or energy efficiency on traditional architectures, dedicated hardware solutions, also referred to as graph processing accelerators, are essential and emerging to provide the benefits significantly beyond what those pure software solutions can offer. In this paper, we conduct a systematical survey regarding the design and implementation of graph processing accelerators. Specifically, we review the relevant techniques in three core components toward a graph processing accelerator: preprocessing, parallel graph computation, and runtime scheduling. We also examine the benchmarks and results in existing studies for evaluating a graph processing accelerator. Interestingly, we find that there is not an absolute winner for all three aspects in graph acceleration due to the diverse characteristics of graph processing and the complexity of hardware configurations. We finally present and discuss several challenges in details, and further explore the opportunities for the future research.

Keywords graph processing accelerator, domain-specific architecture, performance, energy efficiency

1 Introduction

For a wide variety of applications, e.g., date science, machine learning, social networks, roadmap and genomics, the graph is expressive to represent the inherent relationships between different entities. Therefore, graph processing has become a hot topic for solving many real-world problems in both academia and industry. With the growing development of Internet of Things and cloud computing, the size and the complexity of graphs are still expanding. This poses great challenges for modern graph processing eco-systems in both performance and energy efficiency.

There are a large number of studies that attempt to use software solutions to improve the performance and energy efficiency of graph processing. From distributed computing environment[1,2], to single high-end server[3], to the commodity personal computer[4,5], these sys-

Survey

Special Section of NSFC Joint Research Fund for Overseas Chinese Scholars and Scholars in Hong Kong and Macao 2014–2017

This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1003502, and the National Natural Science Foundation of China under Grant Nos. 61825202, 61832006, 61628204 and 61702201.

[*] Corresponding Author

tems basically make tremendous efforts on software optimizations for programmability, high performance and scalability under traditional architectures. In an effort to accelerate graph workloads, multi-core CPUs and GPUs have been recently adopted to expose a high degree of parallelism for high performance graph iteration, e.g., Medusa[6], Cusha[7], GunRock[8], Frog[9], MapGraph[10], and Enterprise[11].

Despite a large number of software solutions, the potentials of graph processing on performance and energy efficiency are still bounded to current hardware architectures. Real-world graphs often follow a power-law distribution in the sense that most of vertices are associated with a few edges, leading to the fact that prohibitive memory access overhead and low efficiency have occurred on general-purpose processors[12−15]. The irregularity in graph processing inherently falls short in exploiting memory- and instruction-level parallelism on traditional processors. It is also observed in the previous studies that a wealth of memory bandwidth is actually under-utilized for graph processing on existing commodity multi-core architectures[15−18].

Though GPUs have demonstrated compelling performance on graph processing[6−8,19], they still suffer from key issues in terms of control and memory divergence, load imbalance and superfluous global memory accesses. More important is that CPUs and GPUs are known for relatively high energy consumption. With the end of Moore's law, using pure software solutions on traditional architectures is often extremely difficult to fill the significant gap between the general-purpose architectures and the graph-specific computation for seeking the top performance of graph processing.

For graph processing, architectural innovation is imperative. Hennessy and Patterson identified the importance, trend and opportunities of Domain-Specific Architecture (DSA) in their recent technical report[20]. It is pointed out that open sourced architectural implementations[①] are the key for the innovations on hardware design[21]. The agile chip development can also shorten the development cycle for DSA prototypes[22]. These guidelines provide one of most effective means for driving the rapid development of graph processing specific accelerators. At this point, hardware platform templates, e.g., Field Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (AISC), are in line with the demand of the times. A large number of industries have already deployed their services on these beneficial hardware platforms for top performance and energy efficiency. For instance, FPGAs have been used in Microsoft datacenter for energy efficiency improvement[23].

Specifically in terms of graph processing, it has been also witnessed that a large number of relevant studies build their graph processing accelerators based on FPGA[24−28] and ASIC[16,29−31]. Evaluation on these accelerators has also demonstrated the efficiency and effectiveness of DSA design[16,28,32].

It is time to review the past and the present of graph processing accelerators, and further look into their future development. In this paper, we conduct a systematic review on graph processing accelerators. It aims at exploring the key issues in the design and implementation of graph processing accelerators. As summarized in Fig.1, we have identified a complete set of core components for graph processing accelerators, which involves three major aspects: preprocessing, graph parallel computation, and runtime scheduling.

- *Preprocessing.* A graph processing accelerator often has limited storage resources. Graphs are needed to be partitioned. Preprocessing is an important component that operates on graph data for trying to make the graph dataset fit into the memory capacity of the graph accelerator. It is also the key to match a certain processing model and appropriate graph representation before the formal processing.

- *Parallel Graph Computation.* The parallel graph computation component serves as the main execution part of graph processing accelerator design. Iterative paradigm is often chosen to define a basic execution pattern for graph iteration that will be mapped to a pipelined hardware circuit. The implementation of this part generally relies on some hardware platform, e.g., FPGA, ASIC, or Processing-In-Memory (PIM). Different specifications have different concerns on hardware designs and sophisticated software co-designs for high throughput and energy efficiency.

- *Runtime Scheduling.* This part aims at how to schedule a large number of graph computational operations on a finite set of hardware resources of graph processing accelerators. The basic metrics for runtime scheduling are to guarantee the correctness and efficiency of graph iteration. The runtime scheduling component often involves data communication, execution mode, and scheduling scheme.

Based on the three aforementioned aspects, we carefully examine the benchmarks and results of existing studies. We find that there is not a clear win-

[①] http://www.riscv.org, Jan. 2019.

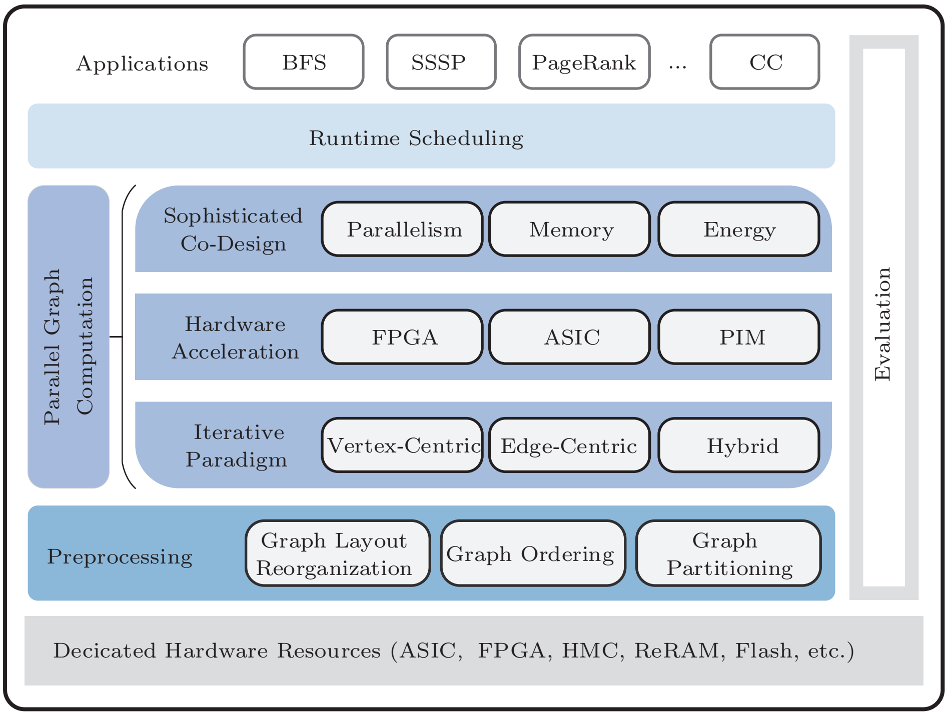


Fig.1. Building blocks for graph processing accelerators (with three major aspects: preprocessing, parallel graph computation, and runtime scheduling).

ner for all these aspects in graph acceleration because of the diverse characteristics of graph processing and the complexity of hardware configurations. We therefore present and discuss several challenges in details, and further explore the opportunities for the future research. One of the major challenges in the existing graph processing accelerators is that the programmability is an important issue for users to express their graph applications. Existing graph processing accelerators typically require labor-intensive efforts for hardware-level modifications.

Great challenges come with great opportunities. Widespread graph applications have a strong demand for energy-efficient graph processing accelerators. Emerging memory devices, e.g., Hybrid Memory Cube (HMC)[33], High Bandwidth Memory (HBM)[34], Resistive Random Access Memory (ReRAM)[35] along with new processing devices, provide us with great opportunities to explore new schemes for graph processing. We believe that this survey summarizes these challenges and opportunities, which can help realize the accelerators with novel hardware-software co-designs.

The rest of this paper is organized as follows. Section 2 includes an introduction to basic components of graph processing, and briefly summarizes the recent progress on CPUs and GPUs. Section 3 presents some considerations in the preprocessing phase. Design and implementation of parallel graph computation are reviewed in Section 4. Section 5 describes the runtime and scheduler part of graph accelerators. Emerging graph accelerators are reviewed and compared in Section 6. Challenges and opportunities are given in Section 7. Finally, this paper concludes in Section 8.

2 Preliminaries

In this section, we first give a brief introduction to the preliminaries of graph processing, including graph representation and several common graph algorithms. Next, we summarize some unique characteristics of graph processing, followed by the related work of graph processing on commodity general-purpose processors. The characteristics of graph processing and the related work further motivate our survey work on graph processing accelerators.

2.1 Graph Representation

Graph is a data structure consisting of vertices that are further associated with edges. A graph can be typically defined as $G = (V, E)$ where V represents the vertex set and E indicates the edge set. For a directed graph, an edge can be represented as $e = (v_i, v_j)$, indicating that there is an edge pointing from v_i to v_j. In particular, a vertex and an edge can be also attributed with a single or multiple attributes. Real-world natural graphs, e.g., social networks, usually have the following three common features.

- *Sparsity.* The average number of vertex degrees is relatively small. The sparsity of graphs can result in poor locality for data accesses.
- *Power-Law Distribution.* A few vertices have associated most of the edges. This can lead to a severe workload imbalance issue with a large number of date conflicts when high-degree vertices are being updated.
- *Small-World Structure.* Two arbitrary vertices in the graph can be connected with only a small number of hops. The small-world feature will make it difficult for partitioning the graph efficiently (as to be discussed in Subsection 3.3).

2.2 Graph Algorithms

We review several common graph algorithms with different requirements in computation, communication, and memory access. These graph algorithms are also widely studied for the exprimental evaluation in the previous studies[12,13,17].

Breadth-First Search (BFS) is a basic graph traversal algorithm, which is used as the kernel of Graph500 benchmarks. The neighboring vertices are iteratively accessed from the root vertex until all vertices of the graph are visited.

Single Source Shortest Path (SSSP) is another graph traversal algorithm that computes the shortest paths from a source vertex to other vertices. Different from BFS, it has less redundant computations in checking edges. Each vertex may be activated more than once. Therefore, it needs more memory space than BFS.

Betweenness Centrality (BC) is widely used to measure the importance of a vertex in a graph. The betweenness centrality value of a vertex is calculated by the ratio of the shortest paths between any other two vertices. The BC algorithm requires to compute the shortest paths between all pairs of vertices.

PageRank is one of the most popular algorithms, which calculates the scores of websites[36]. It maintains a PageRank value for each vertex. All the vertices are activated in each iteration. It often needs large memory bandwidth and float point computing ability.

Connected Components (CC) is widely used in image regions analysis and clustering applications. Each vertex maintains a label. If vertices are in the same connected region, their labels are set to the same. The algorithm updates the labels of all vertices iteratively until converged.

Triangle Counting (TC) is used to measure the number of triangle cliques in the graphs. Each vertex maintains a list of neighbors, and iteratively checks if there are shared neighbors between connected vertices of each pair. The number of triangles is calculated by the overlaps.

Graph Coloring (GC) is to assign colors to the vertices of a graph so that any two adjacent vertices have different colors. GC can be used in many areas, e.g., traffic scheduling, register allocation during compiling and pattern matching. Basic GC algorithm iteratively colors an active vertex with the color that has not been assigned on any of its neighbours.

Collaborative Filtering (CF) is an important machine learning algorithm used for recommendation. Given a bipartite graph where edge values represent the ratings and vertices correspond to the users and items, CF runs iteratively on the bipartite graph to find latent features for each vertex, with all the vertices being active in each iteration.

k-core Decomposition (kCore) is widely used for structure analytics for large cloud networks. This algorithm iteratively removes all the vertices with degrees less than k such that k-core subgraphs are built. Each vertex in a k-core subgraph is with a degree no less than k.

Minimal Spanning Tree (MST) extracts a tree containing all the vertices from an edge-weighted graph with minimum weight. MST is popular in cable network construction, cluster analysis and circuit design. Prim's greedy MST algorithm iteratively chooses the minimum weight edge between vertices in and out of the spanning tree to construct MST.

2.3 Unique Features of Graph Processing

As discussed previously, real-world graphs have the "power-law" distribution and the "small-world" feature. Besides, graph algorithms differ in computational and memory access requirements. Graph processing generally manifests the unique features as follows.

- *Intensive Data Access.* On the one hand, graph applications usually lead to a large number of data access requests. On the other hand, graph processing has a high data-access-to-computation ratio, that is, most of the operations in graph processing are related to data accesses.

- *Irregular Computation.* Due to the power-law distribution, computation workloads for different vertices may vary in a large scale. This will cause severe workload imbalance issue and communication overhead.
- *Poor Locality.* Data accesses of graph processing are usually random because each vertex may connect to any other random vertices. This feature often leads to heavy overhead of memory accesses.
- *High Data Dependency.* The data dependency is caused by the nature of connections of vertices in graph. Heavy dependencies make it difficult to explore the parallelism in graph processing. This may cause frequent data conflicts.

2.4 Brief Introduction to Graph Processing on Modern Commodity Processors

Many graph processing systems have been explored on modern commodity general-purpose processors, e.g., CPUs and GPUs. We briefly introduce the related work to motivate our study, and refer readers to recent surveys for more details[37−39].

- *Graph Processing on CPUs.* There is a large amount of work that aims at building an efficient system for graph applications on CPUs. Basically, they can be divided into two categories. The first kind is the distributed systems[40−45], which leverage the clusters to support massive graph data. However, this usually suffers from communication overhead, synchronization overhead, fault tolerance, and load imbalance issues[46−49]. Emerging servers can hold most of the graph data in the large main memory. Thus, there is an amount of work that exploits the potential of single machine[3,50−52]. There are also many disk-based graph processing systems[4,5,53−56] which can avoid parts of the challenges in the distributed systems. Recently, Many Integrated Core (MIC) architecture based processors have been also explored to improve the performance and efficiency of graph processing[57].
- *Graph Processing on GPUs.* GPU is adopted to pursue high performance of graph processing due to its data parallel capability. A number of graph processing systems with GPUs[6−8,58] have been proposed for high-performance graph processing. Enterprise[11] is developed to accelerate the performance for the BFS algorithm only. There is also plenty of work on accelerating CC algorithm[59], BC algorithm[60,61], and SSSP algorithms[62]. Domain-specific graph processing frameworks have been presented to provide high efficiency for the development on GPUs[63]. To support large-scale graphs, hybrid CPU-GPU systems[64,65], multi-GPUs systems[19,66] and out-of-memory systems[67,68] have been proposed.

Remarks. Despite a significant amount of effort in improving the graph processing performance on general-purpose processors, e.g., CPUs and GPUs, existing graph systems are still far from ideal to exploit the hardware potential of general-purpose processors[15,16]. This is due to a significant gap between the general-purpose architectures and the unique features of graph processing. The graph processing accelerator is necessary as an alternative approach that might be able to fill this gap.

Nevertheless, existing studies on CPUs and GPUs have a wealth of experiences in designing graph accelerators (as discussed in the previous studies[28−30,32]). Various kinds of software graph processing models have been proposed to effectively express graph applications in a generic framework. Partitioning methods, out-of-memory processing and hybrid architectures schemes have been explored to support large-scale graphs.

We next illustrate three aspects of core components of graph accelerators, including preprocessing, parallel graph computation, and runtime scheduling.

3 Graph Preprocessing

The data size of real-world graphs can easily exceed the on-chip/board memory capacity of graph processing accelerators, which is a significant challenge for accelerators. This issue can cause large amounts of I/O and communication cost. In order to make data access efficient, preprocessing of graph data is often required to adapt the data structure onto the target graph accelerators. In this section, we will review the following major graph preprocessing methods used in the designs of graph processing accelerators.

- *Graph Layout Reorganization.* Graph layout is an important factor to affect the graph processing efficiency. Most previous studies have attempted to reorganize the layout to improve data accessing efficiency from many distinct aspects, e.g., data locality, memory storage, and memory access patterns.
- *Graph Ordering.* Graph ordering aims to change the order of the vertices or the edges, such that data locality with less data conflicts can be obtained while the structure of the graph remains the same[27,69].
- *Graph Partitioning.* Graph partitioning is to divide a large graph into multiple disjoint small subgraphs. It usually allows parallel processing of the subgraphs. The processing on each sub-graph has most

of data accesses on the corresponding graph partition. This is particularly useful for improving the cache locality or when the memory of the accelerator cannot hold the entire graph.

3.1 Graph Layout Reorganization

We will introduce the baseline graph layouts first. There are generally two widely-used categories of baseline graph layouts, i.e., edge array and compressed adjacency list. In graphs based on the edge array, each element of the array contains a pair of integers, i.e., source vertex index and destination vertex index. It is convenient to read the edges sequentially from memory. The edge array layout remains widely used in many graph processing systems, especially for the edge-centric processing systems. Another improved edge array layout is Coordinate List (COO). It has been widely adopted in graph accelerators[27,28,70]. It has the edge attributes that are stored along with the edges.

Compressed adjacency list graph originates from the adjacency matrix. It typically uses three arrays to store the graphs, i.e., the vertex property array of the graph, the edge array with the edges' outgoing/incoming vertex indices only, and the edge array starting indices of each vertex in the graph. Suppose outgoing edges are used in the edge array, we name this adjacency list format Compressed Sparse Row (CSR). If incoming edges are used in the edge array, this layout is called Compressed Sparse Column (CSC). The compressed adjacency list graph is relatively compact and beneficial to many graph accelerators[29,71]. Note that the edges of each vertex are stored sequentially.

Based on the baseline graph layouts, we have also many novel methods to compress the data size and optimize memory access further.

- *Combining Information.* Existing work tends to combine multiple information in the same file of graph data layout so that the data locality can be optimized, and random memory access can be reduced.

For instance, [72] proposes to associate the destination vertex property with the edge information such that the vertex property can be sequentially accessed to edges with a good locality. Authors of [25] opted to modify the row pointer array representation in a typical CSR format. They combined the vertex status (1 bit for BFS only) and the vertex's neighboring information in an element of the array. This method improves the memory access efficiency significantly.

- *Encoding Index.* Using an encoding method can compress the graph layout to a small size. Thus, large graphs can be processed on a single accelerator. This is usually done for the index of vertices and edges.

For example, GraphH[73] proposes to squeeze the blank vertex indices by re-indexing the vertices of the graph when the number of vertices is smaller than the maximum vertex index. The index can also be compressed by grouping them with a coarsen ID and using less bits to represent the same graph as presented in [16, 28]. It is also possible to reduce the edge information with frequency-based encoding[74].

Remarks. The baseline graph layouts are useful towards graph accelerators, but they can still be improved for different memory system designs in hardware accelerators. We still have the potential to explore the graph layouts at the aspects of data locality, memory access patterns, and memory footprint.

3.2 Graph Ordering

A number of graph ordering methods have been explored and demonstrated to be effective.

- *Index-Aware Ordering.* It typically targets at the edge array layout. The basic idea is to sort the edges based on either the source vertex indices or the destination vertex indices. Sorting the edges in an ascending manner generally improves the data locality because the neighboring vertex property can be prefetched and probably reused[73]. In the graph processing, source vertex property will be read and destination vertex property will be updated accordingly. Therefore, reading overhead can be reduced if the edges are sorted by source vertices. Similarly, the writing process can be more efficient if the edges are sorted by the destination vertices[27]. As demonstrated in [16, 26, 28], a hybrid index-aware sorting method that balances both the source vertices and destination vertices can outperform the methods that only consider the source vertex or the destination vertex.

- *Degree-Aware Ordering.* This method takes the vertex degree as the sorting metric. Sorting the vertices based on vertex degree in descending order brings multiple benefits[74]. As high-degree vertices are more likely to be accessed, good data locality can be observed if high-degree vertices are placed nearby. In addition, it balances the workloads as well[75] when the graph is processed in parallel. The degree-aware ordering method applies to both baseline graph layouts[76], i.e., the edge array and the compressed adjacency list.

- *Conflict-Aware Ordering.* This method is to reduce the data access conflict during parallel graph processing. ForeGraph[28] proposes to interleave the edges such that memory-level parallelism can be explored more efficiently. Different from the interleaving method, AccuGraph[15] reorders the edges of the whole graph such that the destination vertices of the edges read in each cache line are distributed evenly over the on-chip memory banks. In this case, the parallel destination vertex updating has fewer conflicts.

Remarks. Graph ordering methods focus on changing the order of the graph data organization. The reordered graph can be directly used by the graph accelerators without any modification. Nevertheless, the graph ordering usually requires global sorting and the pre-processing overhead, which can be costly.

3.3 Graph Partitioning

Graph partition makes it possible to fit the graph into the limited on-chip memory of a graph accelerator. The major graph partition strategies in graph accelerator designs can be roughly divided into four categories as shown in Table 1.

Table 1. Partitioning Schemes of Graph Accelerators

Partitioning Scheme	Graph Accelerator
Source-oriented	[15, 27, 69, 77–80]
Destination-oriented	[16, 26, 30, 73, 81]
Grid	[28, 70, 82]
Heuristic	[29, 31, 32, 75, 76, 83, 84]

- *Source-Oriented Partition.* The source-oriented partition methods typically have disjoint source vertices in each partition. All outgoing edges are associated with the partition's source vertices. The destination vertices will be included in the corresponding partition. Particularly, the source vertex indices in each partition are usually continuous to ensure sequential memory accesses. With the source-oriented partition, it is convenient to determine the partitions that need the updated vertex property in the graph processing. Nevertheless, different partitions may be in conflict with destination vertex update. To address this problem, [27] proposes to synchronize through messages and resolve the data dependency through a specific computing unit.

- *Destination-Oriented Partition.* The destination-oriented partition is similar to the source-oriented partition. Basically the partitions have disjoint destination vertices. Therefore, each partition can be updated independently while reading the source vertex property for each partition is mostly random. Graphicionado[16] adopts this partition method to ensure that each partition can be fitted to the small scratchpad memory. Low-latency high-bandwidth scratchpad memory can be fully utilized. GraphP[81] also applies this partition. GraphP[81] aims at reducing the communication between the partitions on different accelerators such that the communication among the HMC cubes can be improved.

- *Grid Partition.* The grid partition of graph in graph processing systems was first introduced in GridGraph[55] which presented an efficient graph data layout and was widely absorbed into designs for graph processing accelerators[28,70]. Grid partition is essentially a two-dimensional partition method, which can be considered as an extension of the one-dimensional partition, like source-oriented partition and destination-oriented partition[28,70]. First, it divides both the source vertices and the destination vertices into continuous segments. Then it forms a two-dimensional array of cubes. Each cube includes the source vertex set, the destination vertex set, and all the edges whose source vertices and destination vertices belong to the source vertex set and the destination set, respectively. The grid partition produces finer grained partitions. The partitions have both sequential source vertices and destination vertices. ForeGraph[28] uses this method to make best use of the limited on-chip memory of FPGAs. In particular, it optimizes the read order of partitions such that the partition loading and processing can be overlapped. This method is also used in GraphR[70] and helps explore the ReRAM features for both high-performance and low-power graph acceleration.

- *Heuristic Partition.* Unlike the above partition methods, many heuristic graph partition methods have been intensively explored, especially for conventional CPU-based graph processing systems. These partition methods follow various heuristic metrics to reduce the communication, to improve locality, or to provide better load balance. Some of them are also applied for the graph accelerator design. For example, a hash-based partition algorithm is used to achieve partitions with balanced vertices and edges in [29]. A clustering-based partition algorithm is adopted for better locality in [76]. A multi-level partitioning algorithm is adopted in FASTCF[75] and is also demonstrated to be efficient for stochastic-gradient-descent-based collaborative filtering.

Remarks. Graph partition brings multiple benefits to graph accelerator design. In particular, it allows the graph accelerator to explore the small yet low-latency high-bandwidth on-chip memory.

Graph preprocessing benefits the graph accelerator on many aspects including better data locality, more efficient memory access patterns, higher task-level parallelism, and even fewer memory accesses. In general, it is a critical step to improve the performance of the graph processing accelerators, and even affects the accelerator design choices. While some preprocessing approaches are extremely time-consuming, it is still an open problem on how to achieve a better balance between the overhead and the performance benefits in many practical scenarios as pointed out in [13].

4 Parallel Graph Computation

The core component of a graph processing accelerator is how to handle the preprocessed graph data in Section 3 with massive parallellism. Considering intertwined data dependencies of graphs, this often requires non-trivial technical innovation, involving matched parallel iterative paradigms, dedicated hardware acceleration and sophisticated co-codesigns. Fig.2 outlines the taxonomy of parallel graph computation.

• *Iterative Paradigm*. Iterative paradigm is used to express the process of how vertices and edges run. It defines the basic data access and computational pattern of graph program. Typical iterative paradigms in existing graph accelerators can be categorized into three approaches: the vertex-centric approach, the edge-centric approach, and the hybrid approach. They decouple the associated dependencies within graphs as many as possible, and further explore the potential parallelism of graph processing.

• *Dedicated Hardware Acceleration*. Different kinds of dedicated hardware platforms can be used to accelerate graph analytics. Existing graph processing accelerators are basically built upon three types of hardware platforms: FPGA, ASIC, and PIM. These emerging architectures can be used to architect efficient memory hierarchy and computing units for higher performance and energy efficiency.

• *Sophisticated Co-Designs*. Sophisticated co-designs usually combine the hardware and the software optimizations to exploit the hardware potentials. They often focus on three aspects: parallelism extension, memory access optimization, and energy efficiency optimization. Most of these co-designs can be commonly used on different kinds of hardware to achieve high performance and energy efficiency.

4.1 Iterative Paradigm

The graph has complex data dependencies between vertices. Designing efficient iterative paradigms is important to decouple these associated dependencies as

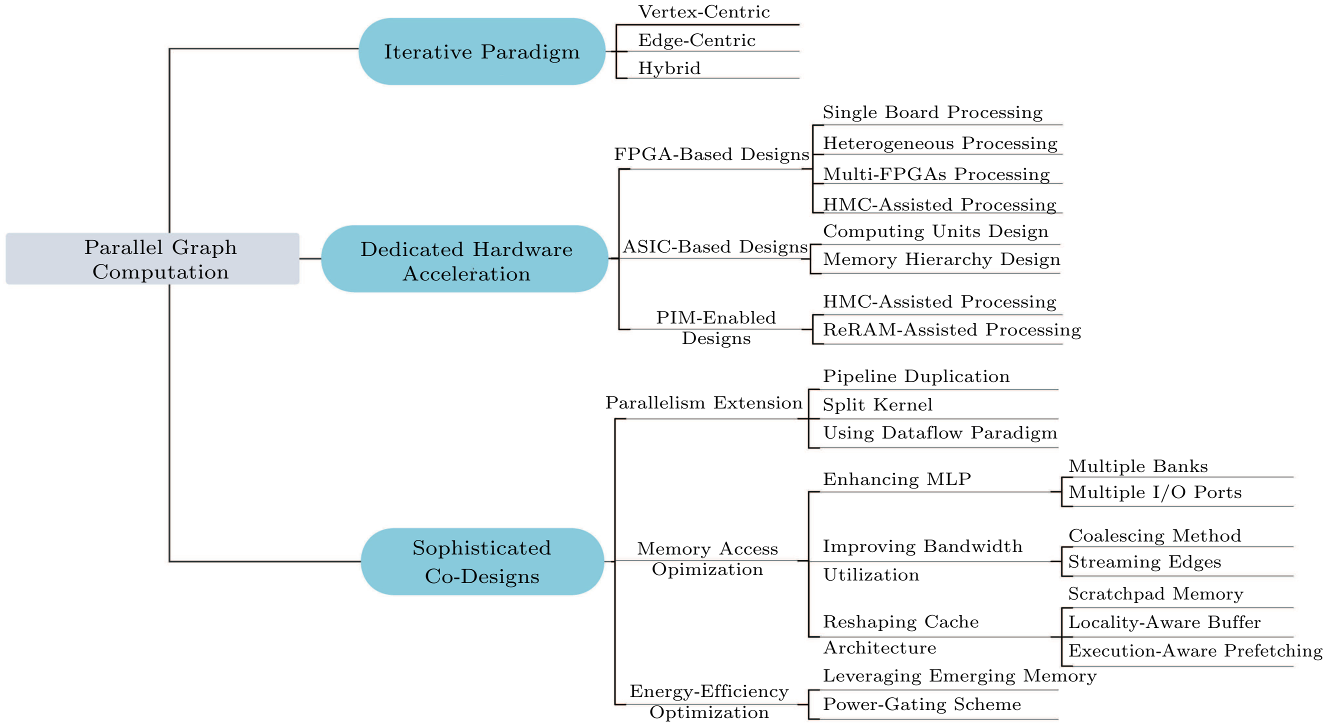


Fig.2. Taxonomy of parallel graph computation.

many as possible by exploring the common computational pattern surrounding vertices and/or edges. Existing iterative paradigms for graph processing can be basically divided into two subcategories: the vertex-centric approach and the edge-centric approach. The vertex- and edge-centric approaches not only concern the expressiveness and abstraction of graph algorithms but also impact the design of graph data layout, preprocessing and computation. A few graph accelerators also have made a hybrid attempt for embracing the best worlds of both modes. Table 2 summarizes the related work with different iterative paradigms.

Table 2. Iterative Paradigms of Graph Accelerators

Iterative Paradigm	Graph Accelerator
Vertex-centric	[14–16, 25, 26, 29-32, 71, 74, 76, 78, 81, 83–91]
Edge-centric	[27, 28, 70, 72, 73, 75, 82, 92–94]
Hybrid	[80]

Programming Model. Programming model is used to effectively express the graph algorithms. It abstracts the common operations in various graph algorithms and alleviates the effort for programmers to write their applications. According to the iterative paradigms, there are vertex-centric programming model and edge-centric programming model. These two models can be combined as the hybrid model to take advantages of both paradigms.

- *Vertex-Centric Programming Model.* Graph algorithms expressed with this model handle the graphs by following "think like a vertex" philosophy[1]. It describes a graph program for each vertex, including computational operations and data transmission between their neighbors via edges. Since each vertex is processed independently, parallelism can be therefore guaranteed by simultaneously scheduling these vertices without data dependencies.

- *Edge-Centric Programming Model.* X-Stream[5] is the first work to use edge-centric programming model to handle graph edges. Unlike the vertex-centric model, this model describes a graph program for each edge. An edge is processed with three steps: 1) collect the information of its source vertices, 2) update its value, and 3) send this value to its destination vertices. It is clear that this model removes the random accesses to edges via sequential streaming of each edge to the chips.

- *Hybrid Programming Model.* Alternative is to use a hybrid method by switching between vertex- and edge-centric programming models for the purpose of taking advantages of both models[80]. The vertex-centric model is responsible for the situation when the active vertex ratio is relatively high. In contrast, the edge-centric model is intended to cope with the case that active vertex ratio is relatively low. Clearly, model switching decision can be made according to the active vertex ratio (among all vertices). The threshold can be decided by the ratio of bandwidth.

Data Layout Selection. Systems implemented in the vertex-centric approach typically iterate over the active vertices and execute the vertex program on them at each iteration. For each given vertex, its neighbours are visited to complete the computation. This kind of implementation usually requires a fast scan for edges of given vertices. As a consequence, as presented in Subsection 3.1, the compressed adjacency lists (CSR/CSC) are suitable for the vertex-centric model because the associated edges of a vertex can be found easily[4,29].

Similar to the edge-centric approach, which iterates over all the edges to implement the edge program for each of the edge, a fast sequential scan of edges is demanded. To process an edge, the information of the end vertices also needs to be indexed directly. Therefore, the edge array presented in Subsection 3.1 intuitively fits for systems in edge-centric approach[5,27].

Preprocessing Considerations. Initially, the graph data is usually stored in the disk as edge files where the edge is represented as a pair of corresponding source and destination vertices. During the preprocessing phase, edge files are converted into the appropriate data layout according to programming models. As discussed in Section 3, preprocessing involves graph partitioning, reorganization and ordering. The complexity of preprocessing also varies for different data layouts.

For the vertex-centric approach, the edge file is converted into the format of adjacency lists. Typically, the edges are sorted by the source or destination vertex followed by index creation that maintains the edge position in the edge array for each vertex. As for edge-centric approach, the edge array is usually loaded directly without specialized data formatting and conversion[5,27]. A detailed research about the cost on preprocessing is presented in [13]. Generally, the preprocessing cost of the vertex-centric approach is higher than the edge-centric one.

Computation Overhead. The vertex- and edge-centric approaches have different computation patterns as discussed before. In the vertex-centric approach, the computation is executed for each vertex which iterates over the neighbors of a given vertex. In the edge-centric approach, the edges are executed as a stream. At this point, the workload characteristics and cache (miss-rate) metrics are significantly different for the two approaches[13].

For workload analysis, the vertex-centric approach supports selective scheduling on only a subset of vertices for each iteration while the edge-centric approach requires a scan of the whole edges, which means that the edge-centric approach induces more computations than the vertex-centric approach.

Cache behaviours are also different between these two approaches. In the vertex-centric approach, the processed vertices can be (locally) cached while it introduces more random accesses by traversing the frontier. In the edge-centric approach, edges can be prefetched for better use of cache, but it causes more random accesses to vertices. Their actual performance may be significantly different, and largely depends on the inherent topology of the graph and features of graph algorithms.

Generally, the vertex-centric approach introduces more random accesses to edges while the edge-centric approach causes more random accesses to vertices. To improve the cache behaviours, optimizations can be applied to these two models, e.g., organizing edge arrays into grids can improve the cache locality[55].

Discussions. Table 3 compares different paradigms from multiple aspects. It is difficult to judge which approach is better because the performance is usually not the same case when different kinds of graph applications are introduced. The authors in [13] made a comprehensive comparison of these two approaches when different approaches and graph algorithms are included.

Vertex-centric paradigm has been widely used to drive many graph accelerators[16,26,29,88], because of its expressive potentials to easily represent various kinds of graph algorithms, and the high parallelism in the grain of vertex. However, in the vertex-centric paradigm, there can be random accesses to edges, resulting in potentially heavy memory access overhead.

Edge-centric paradigm is usually used by existing graph accelerators for improving the utilization of their limited memory bandwidth[27,28,75]. However, the point is that edge-centric paradigm is lack of flexible scheduling potential in contrast to the vertex-centric one. Almost all of edges have to be processed multiple times to complete the whole process. In addition, this paradigm may also lead to a large number of random accesses to vertices. Thus, additional optimizations are often cooperatively needed, e.g., fine-grained partitioning and tailored vertex update strategies[28,70].

For graph processing accelerators, the selection and design of iterative paradigm for graph processing accelerator must also ensure that: 1) programming is easy to use and understand for graph algorithm representation; 2) parallelism is easy to expose and exploit for high throughput and fast hardware development. It is also important to dedicate the accelerators according to the features of applications. Note that advantages can be combined by incorporating hybrid approaches into a design for better performance.

4.2 Dedicated Hardware Acceleration

Existing graph processing accelerators can be built upon various kinds of hardware platforms. Typical hardware accelerators usually adopt only the traditional customized hardware platforms, i.e., FPGAs and ASICs, and have made few modifications on existing computing logic and memory architectures (e.g., DRAM). Some accelerators have re-built their architectures with in-situ computation without excessive data movement, e.g., HMC and ReRAM devices, which is also known as PIM-enabled accelerators. Different hardware configurations have different considerations for performance acceleration. We next review technical advances of these state-of-the-art graph processing accelerators.

4.2.1 FPGA-Based Designs

FPGA is an integrated circuit that enables designers to repeatedly configure digital logic in the fields after manufacturing, also called field-programmable. The

Table 3. Overview of Different Iterative Paradigms

Iterative Paradigm	Programming Model	Data Layout	Preprocessing	Computation Overhead
Vertex-centric	Iterate over vertices	CSR/CSC	Partitioning; ordering; reindexing; higher cost	Frontier-based; random accesses to edges
Edge-centric	Iterate over edges	Edge array/COO	Partitioning; ordering; lower cost	All edges need to be scanned; random accesses to vertices
Hybrid	Mix of vertex- and edge-centric model	Mixed data structures	Sophisticated preprocessing	Model switch

configuration of FPGAs is generally specified via low-level hardware description languages, e.g., Verilog[95] and VHDL[96]. FPGAs are mostly adopted in graph processing accelerators.

Internal Characteristics of FPGAs. There are different kinds of programmable resources on FPGAs, e.g., programmable Logic Element (LE), Static Random Access Memory (SRAM), and Flash and Block RAM (BRAM). However, the fact is that these resources are usually limited to a small number. FPGA can offer high parallelism by architecting these resources with a pipelined Multiple Instructions Single Data (MISD) model. Multiple data can be processed simultaneously at different pipeline stages. Multiple pipelines can be easily duplicated for parallel processing.

Existing Efforts on FPGAs. A graph program is usually designed into a circuit kernel as the basic processing unit according to the programming model (as discussed in Subsection 4.1), which defines the execution pattern[75,87]. These kernels can be easily reconfigured on FPGAs for different algorithms. For building the efficient memory subsystem, a wide spectrum of previous studies make non-trival efforts for the efficient bandwidth utilization of on-chip BRAMs and the off-chip memories. BRAMs provide high bandwidth and low memory latency for randomly accessed vertices. For improving the locality of vertices on BRAM, fine-grained partitioning and dedicated data placement strategies are needed to increase the reuse rate of vertices on BRAM[26,28,74]. As for improving the utilization of off-chip bandwidth, edges can be streamed sequentially from the memory[27].

A number of studies extend to integrate multiple FPGAs into a cluster so as to support large graphs[25,71]. FPGAs with integrated soft-cores are also presented, which can process the graphs in a distributed manner on a single FPGA board with high parallelism[84]. Heterogeneous architectures are also adopted where FPGA and CPU are connected through cache-coherent interconnect. FPGA can access the host memory without the interruption of CPU. These two processors can easily cooperate with each other to process large graphs with higher parallelism than a single FPGA board[80].

There are also a number of studies that aim at exploring the out-of-memory execution on FPGAs for large graphs. The data can be directly streamed from the disks or flashes to the processing units on the FPGA board in these scenarios[26,28]. Recently, Near-Data Processing (NDP) has been cooperatively used to enhance the power of FPGAs for graph processing by off-loading workloads to the integrated HMCs. This provides significantly high bandwidth and parallelism[71,76,97].

4.2.2 ASIC-Based Designs

ASIC is an integrated circuit composed of electrical components, e.g., resistors. It is usually fabricated on a wafer composed of silicon or other semiconductor materials that are customized for a particular use. ASICs are very compact, fast, and low power. Compared with FPGAs, their functions are hard-wired at the time of manufacture. It is not allowed to change the functionality of a small part of the circuit.

ASIC Designs for Graph Analytics. Due to the fixed circuit limitation, ASIC-based graph processing usually utilizes the expressive Gather-Apply-Scatter (GAS) model[40] to form the circuit[29,30]. Each phase is implemented as a hardware module, and runs in parallel with wires that connect different modules. In order to support various graph algorithms, a reconfigurable block can be integrated for users to define the update functions for flexibility.

As for the memory hierarchy, these graph accelerators commonly adopt the scratchpad memory to replace traditional cache. The scratchpad memory acts as a content addressable cache and can be controlled manually. Graphicionado[16] uses eDRAM as the scratchpad memory to store graph data that needs frequent random accesses, e.g., the destination vertices. Dedicated caches of different kinds of graph data are designed in [29] according to the access features. Since these memory resources can be tightly connected to the processing units in an efficient way, ASIC-based graph accelerators can achieve high throughput on the chip.

4.2.3 PIM-Enabled Designs

Different from traditional hardware designs, research on PIM-enabled architectures usually adopts emerging paradigms for achieving impressive performance results by integrating processing units into the memory. It can provide the extremely high bandwidth and low memory access latency with energy saving. The PIM-enabled acceleration is often implemented by leveraging emerging memory devices, e.g., HMC and ReRAM, both of which integrate the in-situ computation in the memory.

HMC-Assisted Graph Processing. HMC has multiple DRAM dies stacked on top of a logic layer that can provide the ability of computation with high memory access parallelism and sufficient instructions for supporting graph processing. As in Fig.3, the DRAM dies are connected via the Through-Silicon-Via (TSV). Storage space in HMC is organized as vaults. The vault is a vertically connected stack of multiple partitions from different DRAM layers. The logic layer is also distributed to different vaults. With multiple DRAM channels for each vault, HMC can provide significantly high memory-level parallelism. HMC can also be easily scaled to consist of a cluster topology of HMCs[98].

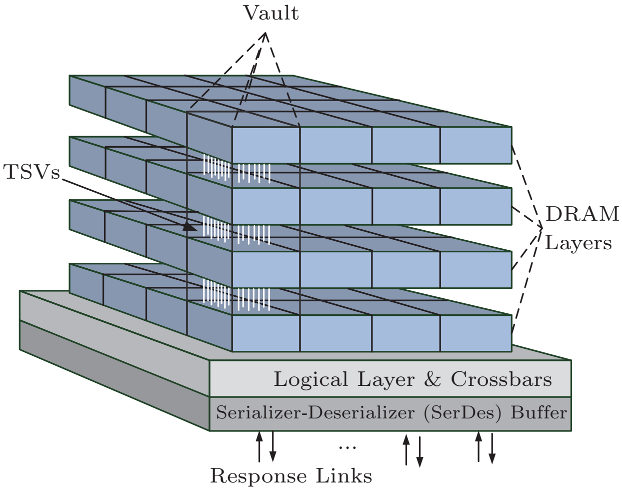

Vault
TSVs
DRAM Layers
Logical Layer & Crossbars
Serializer-Deserializer (SerDes) Buffer
...
Response Links

Fig.3. Illustrative example of HMC architecture.

The logic layer of each vault can work as a softcore with sufficient instruction sets. For better supporting graph processing, instructions have to be reconstructed. Tesseract[32] integrates common instructions of graph algorithms and achieves high performance through multiple HMCs. GraphPIM[14] deigns specialized atomic instructions in HMCs. Besides, graphs are processed in a distributed manner between HMCs. Vertex-cut partitioning is also used to reduce the communication cost between HMCs[73,81].

ReRAM-Assisted Graph Processing. ReRAM is a kind of non-volatile RAM with the enabled computing ability by changing the resistance across a dielectric solid-state material[35]. A ReRAM cell is with high density, low read latency and high energy efficiency[99]. The ReRAM cells can be connected as a dense crossbar architecture to provide high parallelism and memory capacity. Generally, the graph can be represented as a matrix which can be naturally mapped to ReRAM cells. Each cell stores an edge or a vertex. When input voltages are applied to certain rows of the cell arrays, the stored values of each row will multiply the relevant input value. The stored values of each column will be then added together. These features make it possible to realize efficient graph processing on ReRAM.

The potential of ReRAM for efficient computation and storage is still under-studied significantly. To the best of our knowledge, GraphR[70] is the first work to use ReRAM to speed up the graph computation. It transfers the vertex program or the edge program in graph processing to a Sparse Matrix-Vector Multiplication (SpMV) format. However, graph algorithms need to be manually justified for mapping the computational pattern of ReRAM. It is worth noting that there is also a trade-off between the utilization and the throughput due to the limited ReRAM cell size. An ideal situation is that every entity within a matrix is useful for computation for high parallelism. Nevertheless, due to the sparsity of graph data, in a ReRAM block there may be only a few useful edges that are non-zero, causing the fact that a large number of ReRAM cells are underutilized. Extra efforts are still needed to balance this trade-off.

Summary. Considering the reconfigurable feature, FPGA-based designs can handle various kinds of graph algorithms flexibly. FPGA can also provide sufficient interfaces to process large graphs for scale-out efficiency. Massive parallelism can be easily achieved when these resources are in good use. Unfortunately, the resources on FPGAs are limited for existing commodity FPGA boards. The frequency is also relatively low to maintain correctness of execution. These may influence the performance of graph processing.

ASIC designs can provide efficient hardware organizations without the limitation on types and numbers of hardware resources. ASICs can be designed in a relatively efficient way. For example, dedicated and accurate resources placement in ASCIs can be achieved while FPGAs usually have redundant and wasted resources on board. Besides, ASIC can achieve a higher frequency than FPGAs. High performance can be easily attained. However the ASIC chip, once made, is unable to be modified. It is usually difficult for ASICs to handle various graph problems. It is also difficult for ASICs to scale out.

PIM-based accelerators can scale well in both of the bandwidth and the memory capacity. This feature can benefit the graph processing when large graphs are handled. The emerging memories adopted in PIM-based accelerators usually have lower energy consumption than traditional DRAM. To handle generic graph analytics, HMC provides the computing ability by special instruction sets executed in the logic layer. ReRAM processes the graphs in the SpMV format with corssbars. These supports usually need many manual efforts to realize. There is still a lot of research space for PIM-based graph accelerators. For example, the bandwidth can be underutilized due to the communication overhead in HMCs.

4.3 Large-Scale Graph Processing Acceleration

Real-world graph data size can easily exceed the on-chip/board memory capacity of graph processing accelerators. Most of existing accelerators only consider the case that the whole graph fits into the nochip/board memory. However, how to deal with larger graphs on accelerators is a vital issue for practical applications. There is an amount of work that has taken this issue into account, and a series of solutions are further proposed[25,26,28,32,80,94]. These solutions can be typically divided into three categories: the out-of-core solution, the multi-accelerators solution, and the heterogeneous solution.

1) *Out-of-Core Solution.* Unlike traditional CPU architectures that involve large main memory and often develop the out-of-core solutions based on the disks, graph accelerators typically have relatively small on-chip/board memory capacity. Therefore, toward graph accelerators, using any external storages or memories that can store large real-world graphs can be considered potentially-useful as their out-of-core solutions. Graph accelerators can use disks, flashes or other external storage devices to store extremely large-scale graphs[4,5,26,55,94]. However, one of the most important issues for utilizing these devices is to reduce the transmission cost of I/Os between the disk and DRAM since the bandwidths of these devices are often significantly lower than those of DRAM. Streamlined processing schemes[5,94] and sophisticated partitioning methods[26,55] can be explored to effectively reduce the overhead of memory accesses to these external devices. Recently, utilizing embedded processors or accelerators in SSDs has been proved to be another promising way to alleviate the overhead of data transmission and conversion[100−102].

Compared with disk-based solutions, utilizing large host memory enables graph accelerators to process large-scale graphs with better bandwidth-efficiency[27,86,97]. Emerging computing platforms offer the great opportunity for graph accelerators to access the main memory conveniently via specialized interconnections[103]. However, it is also vital to optimize the I/Os between the graph accelerator and the main memory, since long memory latency for data movement often dominates the overall efficiency due to slow I/O interfaces and extra efforts on memory management[104]. Existing memory subsystems and their memory access parallelism are strongly in need of technological innovation.

There also have emerged some studies regarding graph processing accelerator designs for large-scale graph processing. FPGP[26] incorporates the disks to extend the storage of FPGA and designs a streamlined vertex-centric graph processing framework to improve the utilization of the sequential bandwidth of disks. A dedicated on-chip cache mechanism is used to reduce the accesses to disks. Then the large graph is specially partitioned in order to fit for the processing scheme. GraFBoost[94] adopts the flash to scale to much larger graphs and mainly focuses on optimizing the random accesses. The key component is a sort-reduce module that converts small random accesses into large block sequential accesses to the flash storage. It is mentioned that GraFBoost[94] embeds the accelerator into the flash for better scalability. Similar methods have been explored to accelerate the processing in database[105,106]. ExtraV[97] further incorporates the main memory to improve the graph processing with SSDs. Note that host processors can be used together with their self-contained main memory in a heterogeneous solution to enhance the power of graph accelerators.

2) *Multiple Accelerators Extension.* The whole graph needs to be partitioned to distribute different on-chip/board memories of each graph processing accelerator. By considering the prohibitive communication overhead between graph accelerators, the multi-accelerator solution often needs the high-bandwidth connection between graph accelerators. The most important issue for this design is how to achieve a cost-efficient communication mechanism, and avoid data conflicts between graph accelerators. As a consequence, the appropriate graph partition methods are required and are important to reduce the communication overhead[28,32,81]. The inter-network design of graph accelerators is also vital to support the efficient cooperative computing[25,73].

CyGraph[25] runs BFS under a high performance reconfigurable computing platform, Convey HC-2, which constructs a platform with FPGAs connected through a full crossbar to multiple on-board memories. These memories are connected as a shared memory that provides large capacity and high total bandwidth. CyGraph optimizes the CSR representation to reduce the shared memory accesses and connects FPGAs using a ring network to minimize the conflicts. ForeGraph[28] instead uses separated memories for each FPGA. Thus it avoids the memory access conflicts among accelerators. These FPGAs are connected via dedicated inter-connections. Grid-like partitioning[55] and dedicated on-chip data replacement schemes are adopted to achieve better locality for each FPGA board and thus reduce the communications.

As discussed in Subsection 4.2, emerging devices like HMCs not only provide the capability of processing in memory but also scale well. The cost on communications among different HMC cubes dominates the performance[32,73,81]. GraphP[81] utilizes a source-cut partitioning method to significantly reduce the communication overhead. Generally, the multi-accelerator solution is similar to distributed processing under traditional platforms such that many optimizations on distributed graph processing can be applied to accelerators. Meanwhile, the features of different architectures need to be considered to provide the best scenario.

3) *Heterogeneous Acceleration.* As the rapid development of memory integration technologies (e.g., 3D stacking), the host memory becomes large or even huge with trillions of capacity[3,50]. As a consequence, leveraging the host-side memory is alternative to support large-scale graph processing. An intuitive and important question is how graph processing accelerator can interact with the host machine conveniently and efficiently. At present, efficient heterogeneous solutions are still open questions. A few studies propose to use the coherent memory interconnect technology to accelerate graph workloads with CPU and FPGA[80]. For supporting efficient cooperation, the dedicated memory subsystem is needed to alleviate the transmission overhead between the host and the graph accelerator. As a result, the data organization of graphs is the key to reduce the communication overhead. In order to avoid conflicts of computing devices, runtime scheduling schemes are also important for efficient task scheduling.

The authors of [80] proposed to accelerate graph processing under a heterogeneous architecture with CPU and FPGA. Hybrid vertex- and edge-centric models are adopted in [80] as discussed in Subsection 4.1 to fully utilize the processing power of CPUs and FPGAs. Generally, CPU is better for fast sequential processing while FPGA can be used to explore massive parallelism. Hybrid model can flexibly assign workloads to these two devices according to the parallelism of vertices in each iteration. In order to support this scheme, an optimized graph data structure is designed. As for memory coherency, dedicated on-chip memory buffers are designed on FPGA and the accesses to the host memory are controlled by a master thread on CPU. Despite that the heterogeneous solution can extend the power of accelerators, the overhead to maintain the memory coherency might limit the performance. There is still a lot of research space for heterogeneous solutions.

4.4 Sophisticated Co-Designs

Graph processing accelerators often require a series of optimizations for fully exploiting their hardware potentials. There also emerge a few co-optimization techniques at these aspects for high parallelism, lower memory access overhead, and better energy efficiency.

4.4.1 Parallelism Extension

The processing units in either ASIC- or FPGA-based graph processing accelerators are often organized in the form of pipelines. The instructions of graph algorithms are pipelined to offer high parallelism. PIM-based graph accelerators integrate the processing units inside the memory. Their efficiency can be scaled by simply enlarging the memory capacity. For better scalability, three optimization solutions below can be considered useful potentially.

Pipeline Duplication. An intuitive method to increase the throughput is to duplicate multiple pipelines for the parallel processing on more vertices and edges. This simple method has been widely used in a wide spectrum of previous work[16,27,29,30,85,92,107]. Nevertheless, there still remain some potential problems that might prevent the scalable efficiency of multi-pipeline from expectation, which is significantly understudied. For instance, considerable communication between pipelines may lead to the additional overhead via crossbars and controllers[16,29]. In addition, there also exists a workload balance issue that needs specialized data partitioning[16,28].

Split Kernel. Alternative is to split a big, whole processing stream into many small kernels that can be then considered being executed in parallel. This is often done by decoupling the modules of data access and computation, and then making them executed in parallel. In this way, the data access module is responsible for accessing graph data. The computation module uses the data to conduct user-defined computations. For example, by using GAS model, [25, 29, 30] create specialized execution circuits. Each module is thus enabled to process a large number of vertices and edges concurrently. The SpMV-based accelerator[107] also decouples the matrix access from the computation. This method explores the task-level parallelism but extra scheduling mechanisms are needed to ensure the correctness.

Using Dataflow Paradigm. Vertex dependencies of graph can stall the pipelines and decrease the instruction-level parallelism. How to reduce the impact arising from data dependencies remains an open problem for increasing the number of Instructions per Cycle (IPC). One viable solution for solving this problem is to leverage the dataflow paradigm[72,91,108], which forms a directed graph of the operations according to the data dependency between two adjacent operations. The output dependency and the control dependency in graph processing can be then significantly eliminated[91]. GraphOps[72] uses dataflow model to form the data path of different processing blocks. Their overhead of controlling feedback can be therefore alleviated.

4.4.2 Memory Access Optimization

For graph processing, memory accesses often dominate overall execution time. Designing an efficient memory subsystem is crucial for the graph processing accelerator, particularly for memory access efficiency[16,29].

1) *Enhancing Memory-Level Parallelism* (*MLP*). MLP can be represented as the number of outstanding memory requests supported at the same time. Higher MLP can reduce the total memory access time for data-intensive applications as graph processing. It usually needs the memory devices to support enough concurrent memory requests. There are two ways to enhance MLP.

- *Multiple Banks.* One method to increase MLP is using multiple banks. DRAM is composed of many independent banks. Utilizing the parallelism of these banks can significantly improve the memory-level parallelism[85−87]. The memory banks are connected to the processing units directly through crossbars. They can be accessed concurrently.

- *Multiple I/O Ports.* Another method is to design multiple I/O ports for a memory block[27,88,92]. By increasing the I/O ports, multiple memory requests can run concurrently. Usually the number of ports can be manually configured on the scrathpad memory when designing an accelerator. High MLP can be attained when the number of ports is equal to the number of processing units[16]. BRAMs on FPGAs can also be manually controlled to achieve this goal[27]. These BRAMs are usually combined together to form a memory block with multiple I/O ports.

2) *Improving Bandwidth Utilization.* The memory bandwidth utilization here means the valid values ratio per transfer. Random accesses in graph processing usually cause the low ratio of valid values and result in much wasted bandwidth. Improving the memory bandwidth utilization can reduce the total number of memory accesses. There are mainly two effective methods for improving the bandwidth utilization.

- *Coalescing Method.* Coalescing means combining multiple transfers of small items into fewer large ones. This method is widely adopted in graph accelerators[27,71,88,92,93]. For example, if the memory requests are adjacent in a vertex or edge list, these requests can be coalesced as one request for a block. Otherwise there may exist several random accesses that lead to the wasting of bandwidth[88].

- *Streaming Edges.* Streaming edges means that the edges are sequentially accessed from the memory to the accelerator[27]. Random accesses of edges can be reduced. In a vertex-centric model, the edges of a vertex can be streamed to the chip[16]. This method can fully utilize the bandwidth in the edge-centric model. However, the edges may need to be reordered so as to run in a more efficient fashion[27,28].

3) *Reshaping Cache Hierarchy.* Poor locality of graph processing makes the current cache hierarchy lack of efficiency. High cache miss rate will increase the memory access latency, which would cause the underutilization of computing resources. Reshaping the cache hierarchy means designing new cache architectures and mechanisms for graph processing features.

- *Scratchpad Memory.* Scratchpad memory is used as an addressable cache that can be explicitly controlled. The scratchpad memory is closed to the graph engines. It can provide high performance for data access[73,109,110]. Graphicionado[16] uses scratchpad memory to store the temporary vertex property array and edge offset to optimize the random data accesses. Similarly, [29] also designs different kinds of caches for vertices, edges, and other graph information according to their access behaviors.

- *Locality-Aware Buffer.* Locality-aware buffer is a kind of specialized cache for graph data with relatively good locality, e.g., the high degree vertices. High-degree vertices in a power-law graph have higher probability to be accessed many times. These vertices can be cached to improve performance[30]. FPGP[26] and ForeGraph[28] improve the locality of vertices using grid-like partitioning methods, and design special on-chip buffers for vertex subsets, which can be thus accessed fast in reuse.

Execution-Aware Prefetching. This method prefetches the graph data according to the execution requirements. It avoids the inefficiency of fixed traditional cache prefetching mechanism. For example, in the vertex-centric model, the source vertex list and its corresponding edge list can be prefetched sequentially[32]. The key is to exploit the access patterns of different kinds of graph data during the execution, and further design appropriate prefeching mechanism to reduce the memory latency.

4.4.3 Energy Efficiency Optimization

The performance of graph accelerators can be measured as traversed edges per second (TEPS). Energy efficiency can be further defined as TEPS per Watt (TEPS/W). Existing graph processing accelerators can provide significantly high performance by dedicated circuits with inherent low-energy consumption. However, most of graph programs have a high memory-access-to-computation ratio. For example, the energy results show that PageRank consumes over 60% energy on memory[111]. Optimizations on memory consumption can further improve the energy efficiency. Nowadays, there are two simple yet effective ways to improve the memory energy consumption.

Leveraging Emerging Memory Technologies. A number of emerging memory technologies integrate the computing logic inside the memory, e.g., HMC[14,32,73,81] and ReRAM[70,82] as described previously. This architectural reformation can conduct the in-situ computation alongside the data. It naturally avoids the frequent data movement for energy saving. At this point, we can easily replace traditional DRAM by leveraging these emerging memory devices.

Power-Gating Schemes. Power-gating is a widely used technology that powers off the idle logic circuits to save the energy. This scheme is suitable for memories that can be manually controlled[27,82]. For example, it can be applied to BRAMs on FPGAs, which are the key for improving the overall FPGA energy consumption in graph processing accelerators[27]. BRAM is selectively activated and deactivated via the enabled ports. A BRAM module is only activated when the required data is stored. When the edges of a vertex are stored in the same BRAM module, BRAM only needs to be activated once to traverse these edges[27]. Similar strategies can be used for ReRAM[82] to save the energy for edge access by controlling the activation of ReRAM banks in a flexible way.

5 Runtime Scheduling

As discussed in Subsection 4.2, customized hardware circuits for graph processing generally involve specialized designs. This often enforces to design the tailored runtime scheduling to appropriately assign workloads and coordinate the processing units for providing the correct and efficient execution. Unlike existing runtime schedulers on traditional processors, the runtime scheduling for graph accelerators may be necessarily needed to be implemented in the form of hardware circuits. This process usually needs to be transparent to users. Runtime scheduling usually involves three aspects of core components: the communication models, the execution modes, and the scheduling schemes.

- *Communication Model.* Communications commonly exist in graph processing accelerators among processing units. Communication models provide efficient ways for these processing units to communicate and cooperate with each other. Graph accelerators usually adopt two kinds of communication models: the message-based pattern and the shared memory pattern. These models present different features and can benefit from the optimization of information flows.
- *Execution Mode.* The execution mode determines the scheduling order of operations. There are two kinds of execution modes that have been widely used for existing graph processing accelerators: synchronous execution and asynchronous execution.
- *Scheduling Scheme.* The scheduling scheme defines the granularity and processing order of graph data. Existing work adopts three kinds of scheduling schemes: block-based scheduling, frontier-based scheduling, and priority-based scheduling. Flexibly using these scheduling schemes can help reduce the conflicts and improve the utilization of hardware resources.

5.1 Runtime Considerations

For preserving the correctness and efficiency, runtime scheduling for graph processing accelerator needs to consider the following two major aspects.

- *Data Conflicts.* A specific vertex of a graph may be often associated with a large number of edges, particularly true for skewed graphs. There is the common case that some vertices may be updated in conflict by many other vertices simultaneously. For preserving the correctness of vertex updating, the specialized mechanisms are presented to enforce the atomicity. For example, for a read-modify-write update of a destination vertex, [16, 27] propose to use the Content Addressable Memory (CAM) like hardware structure to support finer-granularity memory access, but extra pipeline stalls occur. Similar conflicts can also exist between multiple pipelines. An effective runtime scheduling is expected to avoid these conflicts of vertex updating for high throughput.

- *Workload Balance*. Natural graphs in the real world often manifest the power-law distribution[112]. This can result in a severe load imbalance issue in the sense that a few vertices have extremely high degrees. Workload imbalance may lead to the fact that the loads of some computational logic are overly assigned while other light processing units are stalled. More serious is that the loads of the graph computation are often difficult to predict due to the complex data dependencies. An effective runtime scheduling scheme for graph processing accelerators should be also expected to dynamically balance hardware resources with even loads for every processing unit as many as possible[29].

5.2 Communication Model

The communication model is a well-known pattern that exists commonly to propagate the information between different processing units. We next survey several patterns that have been used in off-the-shelf graph accelerators.

Message-Based Pattern. Message-based communication model is widely used in distributed environments. In message-based communication model, communication is realized by sending messages among different processing units. These massages can carry the updated data or computation commands that will be executed locally. This model is widely used in HMC-assisted graph processing accelerators[32,81]. As mentioned previously, the vaults in HMCs communicate with one another via messages.

Tesseract[32] designs the remote function call mechanism via message passing to indicate the running of destination processing cores. The message passing can be used to avoid the cache coherence issues of the processing cores. It can also reduce the atomic operations for shared data. However, a large number of messages come with a high cost of communication time and bandwidth. Partitioning methods and coalescing methods are usually needed to reduce the number of messages[81]. Besides, extra memory copying operations and buffers are also needed.

Shared Memory Based Pattern. The shared memory model is suited for the communication between processing units on a single accelerator. The same location of a memory can be accessed and updated by multiple processing units simultaneously. When multiple accelerators are adopted, it is also possible to have a distributed shared memory.

FPGP[26] adopts this model based on FPGAs. It maintains a global shared vertex memory for multiple FPGA boards and each board keeps a vertex cache for multiple processing units. Synchronization between iterations is needed to maintain memory consistency. Constrained by limited bandwidth, the global shared vertex memory can limit the scalability of FPGAs. ForeGraph[28] uses a distributed shared memory. Shared memory model can usually avoid the redundant copies of graph data and extra storage space in message passing model. It is also easy to implement and design. However, there may exist many data races on the same memory location if some vertices are updated by many neighboring vertices.

5.3 Execution Model

The execution model typically has two major concerns: 1) scheduling timing, and 2) scheduling order. The scheduling timing indicates when to execute the vertex programs, which can be often synchronous or asynchronous. The scheduling order indicates the information flow for a vertex program to decide how to update the vertex. They are often used to co-determine when and how a vertex can execute an update if it is active.

Synchronous Mode. In the synchronous execution mode, all the vertices in a graph are processed in certain order during each iteration. Between two consecutive iterations, there is a global barrier to ensure that all the newly updated vertices in current iteration are visible at the same time in the next iteration for all processors[113]. In graph accelerators, the graph is usually partitioned into subgraphs that are processed by different processing units. When a processing unit finishes its work, it has to wait for other processing units finished. Then the values of different subgraphs are synchronized[25]. During each iteration, only the local values of graph data can be accessed and updated[26].

The synchronous execution is easy to realize on graph accelerators and suits for memory-bound graph algorithms. It can utilize the memory bandwidth better because the data is updated in a bulk synchronous way. Many memory accesses can be combined and sequential. However, as discussed before, the synchronous mode may require more storage space for local data in each iteration when workloads are unbalanced.

Asynchronous Mode. In the asynchronous execution mode, each processing unit can start the next iteration immediately when it finishes current workloads. There is no global barrier to synchronize these processing units. The asynchronous mode can be used to balance the loads because the processing units are kept busy nearly all the time. This mode suits for the algorithms that converge faster than synchronous execution. Some graph algorithms can only converge under asynchronous execution, e.g., the graph coloring algorithm. It also supports dynamic scheduling, e.g., the priority-based scheduling mechanism[29] to achieve high performance. However, the disappointing point is that the asynchronous mode requires tremendous efforts to implement on graph accelerators for the sophisticated hardware design[114].

Information Flow Direction. For executing a vertex program, it is important to decide how to update the value of vertices. The information flow between vertices typically has two kinds of directions: the push-based mode and the pull-based mode. For an active vertex, the information is propagated from the active vertex to its neighbors in the push mode, while in the pull mode the information is flowed from its neighbors to the active vertex. For the BFS algorithm, in the push mode, the values of out-degree neighbors are updated according to active vertices. In the pull mode, the active vertex gets information from its in-degree neighbors to update itself.

Usually, the push mode can explicitly select the update vertices but it may cause redundant random accesses when seeking the next frontier. Locks might be needed to ensure the consistency since a vertex may be updated by multiple active vertices. The pull mode presents better locality for updated vertices and has natural consistency because the vertices just update themselves. However, it may result in additional overhead for checking whether the updating of neighboring vertices is necessarily executed.

Push and pull modes can be also combined together and switched at runtime to alleviate the synchronization and communication overhead[115]. Ligra[3] first adopts this method into shared memory graph processing systems, and Gemini[45] is the first to apply this hybrid mode to a distributed memory setting which achieves extremely high performance. This hybrid method has also been used in some graph accelerators for performance improvement[74,87]. The switching time is based on the number of active vertices in the frontier and associated unexplored edges. We can switch to the pull mode if the frontier has a high ratio of the unexplored edges for better performance[74].

5.4 Scheduling Schemes

There are many runtime scheduling schemes that can be adopted in graph processing accelerators.

- *Block-Based Scheduling.* In block-based scheduling, the whole graphs are evenly partitioned into blocks and are distributed to multiple processors. There is no strict order for these partitions to be processed. This scheduling method is widely used for graph processing integrated with multiple accelerators.

For example, Tesseract[32] distributes the graphs to multiple vaults on HMCs to process in parallel. ForeGraph[28] partitions the graph into a grid and distributes the grid blocks to different FPGA boards. These executions of subgraphs are usually synchronized after each iteration. The batch-based scheduling can easily help achieve massive parallelism among multiple accelerators in a synchronous fashion. However, the workloads of each batch should be balanced to achieve better resources utilization.

- *Frontier-Based Scheduling.* This kind of scheduling is suitable for those graph algorithms in which only a subset of data needs to be processed in each iteration. A frontier is needed to contain the active data that is to be scheduled. For example, in the vertex-centric model, the frontier contains the active vertices that need to be executed for each iteration. The scheduler gets a vertex from the frontier and checks the state array to decide the data path of the vertex[30,86,114]. The frontier-based scheduling can help process most of graph algorithms. However, the frontier might be modified frequently by multiple vertices which contend for updating the same vertex with serious race conditions. The specialized hardware circuit design may be a viable solution for efficiently supporting multiple simultaneous updates.

- *Priority-Based Scheduling.* In the priority-based scheduling, the scheduled items are assigned a priority flag which represents the execution order. This kind of scheduling is usually combined with the frontier-based approach where the active vertices are ranked. It can also be used to schedule the order of messages to be processed[32]. Prioritiy-based scheduling can help some graph algorithms converge faster in a asynchronous execution model, e.g., the PageRank algorithm[29].

For example, a specialized synchronization unit is designed in [29] to rank and schedule active vertices. These active vertices are maintained in an active list, and they are then executed according to the ranking value. However, the newly created dependencies based on the priorities may bring extra synchronization overhead. Fortunately, the latency can usually be compensated by the gains because of the fast convergence.

Remarks. A single graph processing accelerator may have limited hardware resources and memory capacity. For mobilizing the potentials of these resources, in addition to the effective resource layout, an efficient runtime scheduling scheme is the key, which decides when and where a specified data is supposed to be processed. Considering the complexity of the hardware circuit layouts, unlike the pure software implementations, the runtime scheduling on a graph accelerator has to be co-designed with the necessary hardware supports in many cases for better efficiency.

For instance, software-assisted runtime scheduling for ensuring the sequential consistency can use locking mechanisms that are easy to implement. However, these mechanisms can be also error-prone and even suffer from significant performance degradation in hardware implementation. The specialized hardware supports with CAM structure[109] or more advanced designs[15] make the scheduling for sequential consistency easy. Runtime scheduler can therefore focus more on the parallelism exploitation[114]. In addition, this also greatly mitigates the atomicity overhead. Combined with irregular accesses and large sizes of graphs, more extra efforts still have to be done for runtime scheduling.

6 Graph Accelerator Evaluation

The key issues of the design and implementation of graph accelerators have been summarized in Section 3, Section 4, and Section 5. These designs differ in preprocessing methods, programming models, and hardware architectures. Here we summarize the key metrics in existing work and make a detailed discussion from following aspects.

- *Evaluation Metrics.* Evaluation metrics presented in this paper include the typical design techniques, hardware platform parameters, performance metrics, and energy efficiency metrics. These metrics provide an overall view of different graph accelerators.
- *Summary of Results.* Based on the evaluation metrics, we analyze these results and make a discussion from five aspects: graph benchmarks, platform parameters, preprocessing, graph processing frameworks, and programming models. Various kinds of graph benchmarks and platforms make a fair comparison of different accelerators difficult. Different kinds of design methods can also influence the performance. We argue that it demands standard graph accelerator benchmarks for efficient evaluations.
- *Case Study.* In the review, we find that there is no absolute winner among existing graph processing accelerators in terms of performance and energy efficiency. In this section, we choose another angle to study the design and implementation of a state-of-the-art accelerator[15] in more depth so that readers can have a more in-depth understanding on the three core components.

6.1 Evaluation Metrics

In order to assess the graph accelerators, existing work typically uses TEPS as the performance metric, TEPS/W or power consumption (watt, or joule per read/write) as energy efficiency metric. These metrics basically give an overall evaluation of the graph acceleration system.

Key parameters of existing graph accelerators for evaluation are divided into three aspects. Table 4 gives an overview of a graph processing accelerator including the pre-processing, programming models, and compared systems. Note that each study is assigned with a unique ID which is also used for the same accelerator. Table 5 summarizes the hardware parameters of graph accelerators. Table 6 summarizes the comparison of performance and energy efficiency reported in the related work.

For fidelity, the labels “M” and “S” are used to distinguish the measurement-based results and the simulation-based results respectively in Table 5. We try to provide the actual performance/energy metrics, but some related work has only the relative performance/energy over the compared systems. We thus cannot infer the actual accelerator performance according to their original results. In this case, the performance/energy is labeled as “SP” (speedup) in Table 6. Some accelerators support only a single graph algorithm or a few graph algorithms. The corresponding performance will be labeled as “-”. In addition, we use

Table 4. Overview of Graph Processing Accelerators

Year	System	Architecture	Data Layout	Preprocessing	Programming Model	Generality	Scheduling	Compared System	ID
2016	Graphicionado[16]	ASIC	COO	Y	V/Sync	Various	F	GraphMat[51]	1
2016	EEA[29]	ASIC	CSR	Y	V/Async	Various	P	Host	2
2017	TuNao[30]	ASIC	COO	Y	V/Async	Various	F	Cusha[7]	3
2017	GAA[83]	ASIC	CSR	Y	V/Async	Various	P	Host	4
2018	Ayupov *et al.*[31]	ASIC	CSR	Y	V/Async	Various	P	GAP[116]	5
2015	Tesseract[32]	PIM	-	Y	V/Sync	Various	B	Host	6
2017	GraphPIM[14]	PIM	CSR	N	V/Sync	Various	F	GraphBIG[17]	7
2017	RPBFS[69]	PIM	CSR	Y	-/Sync	BFS	B	Enterprise[11]	8
2018	GraphR[70]	PIM	COO	Y	E/Sync	Various	B	GridGraph[55]	9
2018	RPBFS[77]	PIM	CSR	Y	-/Sync	BFS	B	Enterprise[11]	10
2018	GraphP[81]	PIM	-	Y	V/Sync	Various	B	Tesseract[32]	11
2018	GraphH[73]	PIM	COO	Y	E/Sync	Various	B	Tesseract[32]	12
2010	Wang *et al.*[78]	FPGA+SoC	CSR	Y	V/Sync	BFS	F	Cell BE[117]	13
2011	Betkaoui *et al.*[85]	FPGA	CSR	N	V/Sync	GC	B	GraphCrunch[118]	14
2012	Betkaoui *et al.*[86]	FPGA	CSR	N	V/Sync	BFS	B	PACT11[119]	15
2012	Betkaoui *et al.*[87]	FPGA	CSR	N	V/Sync	APSP	B	HPCC11[120]	16
2014	GraphGen[88]	FPGA	COO	Y	V/Sync	Various	F	Host	17
2014	CyGraph[25]	FPGA	CUST	Y	V/Sync	BFS	F	ASAP12[86]	18
2015	Attia *et al.*[89]	FPGA	CUST	Y	V/Sync	APSP	F	BGL[121]	19
2015	Umuroglu *et al.*[79]	FPGA+SoC	CSC	Y	-/Sync	BFS	F	Host	20
2015	Zhou *et al.*[92]	FPGA	COO	Y	E/Sync	SSSP	B	CyGraph[25]	21
2015	Zhou *et al.*[93]	FPGA	COO	Y	E/Sync	PageRank	B	Host	22
2015	GraphSoC[84]	FPGA+SoC	-	Y	V/Sync	Various	B	Host	23
2016	FPGP[26]	FPGA	COO	Y	V/Sync	BFS	B	GraphChi[4]	24
2016	GraVF[90]	FPGA	-	Y	V/Sync	Various	B	-	25
2016	GraphOps[72]	FPGA	CUST	Y	V/Sync	Various	F	X-Stream[5]	26
2016	Zhou *et al.*[27]	FPGA	COO	Y	E/Sync	Various	B	X-Stream[5]	27
2016	SpMV[107]	FPGA	-	N	-/Sync	SpMV	B	Host	28
2017	ForeGraph[28]	FPGA	COO	Y	E/Sync	Various	B	FPGP[26]	29
2017	Ma *et al.*[122]	FPGA	-	N	-/Async	Various	B	Host	30
2017	Zhang *et al.*[71]	FPGA	CSR	Y	V/Sync	BFS	F	FPGP[26]	31
2017	Zhou and Prasanna[80]	FPGA+CPU	CUST	Y	Hybrid/Sync	Various	F	GraphMat[51]	32
2018	Zhang and Li[74]	FPGA	CSR	Y	V/Sync	BFS	F	FPGA17[71]	33
2018	Khoram *et al.*[76]	FPGA+HMC	CSR	Y	V/Sync	BFS	F	FPGA17[71]	34
2018	FASTCF[75]	FPGA	COO	Y	E/Sync	CF	B	SIGMOD14[18]	35
2018	Yao *et al.*[15]	FPGA	CSR/CSC	Y	V/Sync	Various	F	ForeGraph[28]	36
2018	GraFBoost[94]	FPGA+Flash	CSR	Y	E/Sync	Various	B	FlashGraph[123]	37

abbreviations for some long terminologies because of the limited space. In programming model category, we use "V" and "E" to represent the vertex-centric model and the edge-centric model, respectively. When the model is not clearly named, we use "-" instead. Similarly, we use "Sync" and "Async" to represent the synchronous execution and the asynchronous execution, respectively. Block-, frontier- and priority-based scheduling methods are represented by "B", "F", and "P", respectively.

6.2 Summary of Results

We analyze the summary in the following aspects, including graph benchmark, platform parameter, preprocessing, graph processing framework, programming models, and runtime scheduling.

1) *Graph Benchmark.* When comparing the accelerators, the benchmark is of vital importance to understand the effectiveness of the design and the implementation of a graph processing accelerator. A graph benchmark consists of at least four aspects including graph layouts, types of input graphs, the size of the graphs, and graph algorithms. As shown in Table 4, graph layouts are different across the existing studies on graph processing accelerators. Thus, in fact it requires further research for developing a fair and practical benchmark for evaluating different graph processing accelerators. Particularly, we have the following observations for further research.

First, existing studies use different storage layouts.

Table 5. Parameters of Graph Accelerator Platforms

ID	Compute Device	Frequency	On-Chip Memory	Off-Chip Memory	Bandwidth	Method
1	Streams × 8	1 GHz	eDRAM 64 MB	DDR4 × 4	68 GB/s	S
2	AU × 4	1 GHz	Cache 34.8 KB	DDR4	12.8 GB/s	S
3	ECGRA	300 MHz	Cache 2.4 MB	-	288 GB/s	M
4	AU × 4	1 GHz	-	DDR4	12.8 GB/s	S
5	AU × 4	1 GHz	-	DDR4	12.8 GB/s	S
6	HMC (512 cores)	2 GHz	Cache 16 MB	HMC1.0 × 16	8 TB/s	S
7	CPU (16 cores)	2 GHz	Cache 16 MB	HMC2.0	480 GB/s	S
8	ReRAM (1 024 × 1 024)	1.2 GHz	eDRAM 4 MB	ReRAM	50 GB/s	S
9	ReRAM (32 × 64)	-	ReRAM	Disk	-	S
10	ReRAM (1 024 × 1 024)	1.2 GHz	eDRAM 4 MB	ReRAM	50 GB/s	S
11	HMC (512 cores)	1 GHz	Cache 49 MB	HMC2.1 × 16	5 TB/s	S
12	HMC (512 cores)	1 GHz	SRAM 576 MB	HMC2.1 × 16	5 TB/s	S
13	Virtex-5 FPGA	100 MHz	BRAM 1.29 MB	DDR3	0.1 GB/s	S
14	Virtex-5 FPGA × 4	75 MHz	BRAM 5.18 MB	-	80 GB/s	M
15	Virtex-5 FPGA × 4	75 MHz	BRAM 5.18 MB	-	80 GB/s	M
16	Virtex-5 FPGA × 4	75 MHz	BRAM 5.18 MB	-	80 GB/s	M
17	Virtex-6 FPGA	100 MHz	BRAM 1.87 MB	DDR2	6.4 GB/s	M
18	Virtex-5 FPGA × 4	75 MHz	BRAM 5.18 MB	-	80 GB/s	M
19	Virtex-5 FPGA × 4	75 MHz	BRAM 5.18 MB	-	80 GB/s	M
20	FPGA/ARM	150/666 MHz	BRAM 0.56 MB	DDR3	3.2 GB/s	M
21	Virtex-7 FPGA	200 MHz	BRAM 4.5 MB	DDR3	20 GB/s	M
22	Virtex-7 FPGA	200 MHz	BRAM 8.375 MB	DDR3	20 GB/s	S
23	ZC706 FPGA/SoC	250 MHz	BRAM 70 KB	DDR3	-	M
24	Virtex-7 FPGA	100 MHz	BRAM 4.76 MB	DDR3	12.8 GB/s	M
25	Virtex-7 FPGA	150 MHz	BRAM 6.6 MB	DDR3	-	M
26	Virtex-6 FPGA	150 MHz	BRAM 4.76 MB	DDR3	38.4 GB/s	M
27	Virtex UltraScale FPGA	250 MHz	BRAM 12.8 MB	DDR4	19.2 GB/s	S
28	FPGA × 4	-	-	DDR3 × 8	102.4 GB/S	M
29	Virtex UltraScale FPGA	200 MHz	BRAM 16.61 MB	DDR4	19.2 GB/s	S
30	Virtex UltraScale 440 FPGA × 2	200 MHz	BRAM 22 MB	DDR3	51.2 GB/s	S
31	AC-510 FPGA	125 MHz	BRAM 4.75 MB	HMC2.0	60 GB/s	M/S
32	Arria10 FPGA/ Xeon-cores × 14	-	BRAM 6.6 MB	DDR3	12.8 GB/s	M
33	AC-510 FPGA	125 MHz	BRAM 4.75 MB	HMC2.0	60 GB/s	M/S
34	AC-510 FPGA	125 MHz	BRAM 4.75 MB	HMC2.0	60 GB/s	M
35	Virtex UltraScale+FPGA	150 MHz	RAM 43.3 MB	DDR4 × 2	38.4 GB/s	M
36	Virtex Ultrascale+FPGA	250 MHz	BRAM 9.49 MB	DDR4	19.2 GB/s	S
37	VC707 FPGA/Flash	125 MHz	BRAM 4 MB	DDR3	10 GB/s	M

Some of them adopt the edge list, some of them use CSR/CSC, and some of them utilize the customized layout (CUST). They affect the memory access patterns dramatically and the performance accordingly.

Second, according to Table 6, the types of the graphs used in the accelerators are not totally the same. Types of graphs used in prior work include real-world graphs, e.g., social network graph, road network graph, and functional magnetic resonance imaging (fMRI) graphs. There are also synthetic graphs, i.e., the recursive matrix (RMAT) graph, the Kronecker graph, the graphs generated by the Linked Data Benchmark Council (LDBC), and the graphs generated by the Library of Efficient Data Types and Algorithms (LEDA). Different combinations lead to diverse results.

Third, graph algorithms used in different graph accelerator designs are also usually different. If the algorithms used are different, comparing the metrics of performance and energy efficiency needs to be improvable and justified.

Fourth, graph size is another key graph parameter, but it is not sufficiently considered in previous

Table 6. Comparison of Performance and Energy Efficiency

ID	BFS	SSSP	PageRank	SpMV	Energy	$\lvert V\rvert_{\max}$	$\lvert E\rvert_{\max}$	Dataset
	(GTEPS)	(GTEPS)	(GTEPS)	(GTEPS)	Efficiency	(Million)	(Million)	Type
1	0.125–2.6	0.25–2.3	4.5–4.75	-	7 W	61.570 0	1 468.360	Social/RMAT
2	-	SP	SP	-	3.375 W	67.000 0	1 000.000	Social/Kronecker
3	SP	SP	SP	SP	9.6 W	7.400 0	192.000	Social
4	-	SP	SP	-	SP	67.000 0	1 000.000	Social/Kronecker
5	-	SP	SP	-	SP	16.800 0	268.000	Social/Kronecker
6	-	SP	SP	-	94 mW/mm2	7.400 0	194.000	Social
7	SP	SP	SP	-	-	1.000 0	28.800	LDBC
8	0.2–1.2	-	-	-	-	2.390 0	7.600	Social
9	SP	SP	SP	SP	1.08 pJ(r), 3.91 nJ(w)	4.800 0	106.000	Social
10	0.2–1.2	-	-	-	1.59 pJ(r), 5.53 nJ(w)	1.960 0	5.530	Social
11	SP	SP	SP	-	SP	4.800 0	6.900	Social
12	SP	-	320–350	-	133 mW/mm^2	41.700 0	6 640.000	Social
13	0.16–0.79	-	-	-	-	0.064 0	1.024	Synthetic
14	-	-	-	-	-	0.300 0	3.000	LEDA
15	0.25–2.6	-	-	-	-	16.000 0	1 024.000	RMAT
16	-	-	-	-	-	0.038 0	-	fMRI
17	-	-	-	-	-	0.110 0	0.340	Image
18	1.68–2.2	-	-	-	-	8.000 0	512.000	RMAT
19	-	-	-	-	-	0.065 0	4.190	RMAT
20	0.09–0.255	-	-	-	-	2.000 0	67.000	RMAT
21	-	1.6	-	-	-	1.000 0	-	RMAT
22	-	-	0.27–0.38	-	-	2.390 0	7.600	Social
23	-	-	-	0.015	-	0.017 0	0.126	SpMV
24	0.01–0.012	-	-	-	-	1 400.000 0	6 600.000	Social
25	3.5	-	3	-	-	0.002 5	0.010	Synthetic
26	-	-	0.035–0.115	0.2–0.75	-	2.390 0	30.600	Social
27	-	0.657–0.872	-	-	19.06–24.22 W	4.700 0	65.800	Social
28	-	-	-	0.316	2 MTEPS/W	-	-	-
29	0.897–1.458	-	0.997–1.856	-	-	1 410.000 0	6 640.000	Social
30	SP	SP	-	-	5–8 W	24.000 0	64.000	Synthetic
31	0.13–0.166	-	-	-	-	33.500 0	536.800	RMAT
32	0.33–0.67	0.063–0.075	-	-	-	10.000 0	160.000	RMAT
33	0.4–152.6	-	-	-	43.6 W	23.900 0	577.100	Social/RMAT
34	0.1–0.65	-	-	-	-	16.000 0	252.800	Social
35	-	-	-	-	13.8 W	1.300 0	460.000	Bipartite
36	1.5–3.5	-	1.25–2.5	-	-	3.070 0	117.000	Social
37	0.057–0.075	-	SP	-	50 W	3 000.000 0	128 000.000	Social/Kronecker

work. The graph size used in different graph accelerators varies in a large range as the maximum number of vertices $|V|_{\max}$ and edges $|E|_{\max}$ presented in Table 6. Some graphs have less than a million vertices while some of them have more than a billion. Given even the same type of graph algorithms, the graphs can involve different sizes, especially the RMAT graphs. The number of vertices or edges may vary according to the configuration of the graph generator. As a result, different average degrees of graphs can result in distinct parallelism and data locality of vertices. Therefore, this may lead to different performance in the end.

2) *Platform Parameter.* We find that, even with the same hardware component design, existing graph processing accelerators have different parameter settings. According to Table 5, it is clear that the platforms, i.e., ASIC, PIM and FPGA used in different accelerator designs, make a big difference on the resulting performance and energy efficiency. This is expected since the implementation frequency may have already been different in an order of magnitude.

However, the parameters of the same kind of platform also vary dramatically. For instance, the largest FPGA on-chip memory is around 44 MB while the smallest one is only 0.25 MB. Similarly, the memory bandwidths of the same type of platforms also differ significantly. Large memory bandwidth allows more parallel processing. Large on-chip memory improves the memory access efficiency. The platform parameters can have considerable influence on performance and energy efficiency.

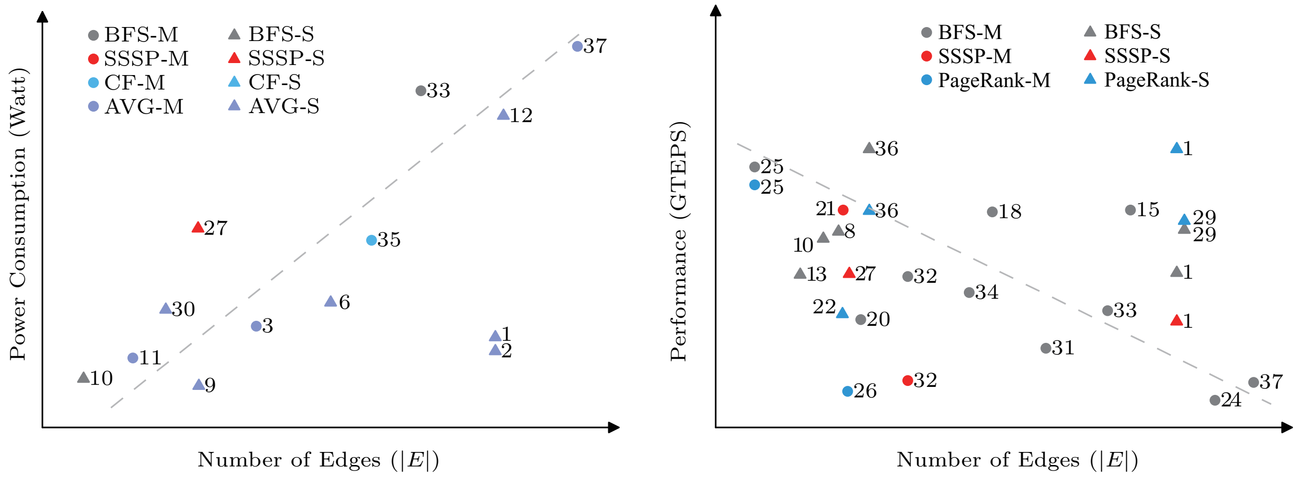
3) *Preprocessing.* As discussed in Section 3, preprocessing is usually beneficial to graph processing as it improves the data locality or memory access patterns. While we notice that some graph processing accelerators do not involve preprocessing at all, it is unfair to make an end-to-end comparison to the ones with preprocessing. In addition, the accelerators with preprocessing can also have diverse preprocessing efforts. When the preprocessing efforts are different, it is also tricky to compare the accelerators. In some of the occasions, when the preprocessing cost can be fully amortized, we may just ignore the preprocessing overhead. It may not be the case when the application is sensitive to preprocessing cost as suggested in [13].

4) *Graph Processing Framework.* According to the "generality" column in Table 4, most of the graph processing accelerators target a set of typical graph processing algorithms, while the other accelerators may focus on optimizing a specific graph processing algorithm. It is essentially a trade-off between generality and performance. It is not fair to compare these accelerators when "generality" is different.

5) *Programming Model.* From the tables, it can be found that different programming models are used in the graph processing accelerators. The accelerators can be implemented in either the synchronous model or the asynchronous model. Also, some accelerators follow a vertex-centric processing model while others choose the edge-centric model. Note that there is also one graph accelerator based on the hybrid model. Different models may also influence the performance of graph accelerators. Nevertheless, there is no clear difference in terms of the ease of programming. Different from the above parameters, accelerators with different programming models remain comparable.

6) *Development Trend.* For further exploration of the results, Fig.4 makes a qualitative analysis of the relative development trend. These two charts only present the relative position of the results for a quick evaluation. More explicit details can refer to Table 6.

Fig.4(a) depicts the relative energy efficiency (represented in power consumption) of investigated graph processing accelerators as the graph size increases. Fig.4(b) illustrates the relative performance of the investigated graph processing accelerators for BFS, SSSP and PageRank with different graph sizes. The graph size is measured by the largest number of edges in respective literature because the number of edges is usually much larger than the number of vertices in the datasets. Edge numbers are depicted in the format of offset reciprocal. The power consumption and performance are depicted in a logit format for qualitative comparison. The ID number of each graph processing



(a)

(b)

Fig. 4. Relative development trend of (energy efficiency and/or performance) results for existing state-of-the-art graph processing accelerators (explicit results can refer to Table 6 for details). "-M" represents the measurement-based results and"-S" represents the simulation-based results. (a) Relationship of energy efficiency and graph size. (b) Relationship of performance and graph size.

accelerator is labeled besides corresponding accelerator's data point in Fig.4. Note that all the data is based on the explicit descriptions in relevant literatures, and the measurement-based results are distinguished from simulation-based results for the fidelity.

Power consumption is an important metric to measure the energy efficiency[29]. The power consumption in Fig.4(a) presents an increasing trend as the graph size increases. This is because it generally demands more computing and storage resources to handle large graphs. Besides, different kinds of hardware designs can contribute to various energy behaviours. The accelerator with the lowest power consumption adopts the emerging ReRAM which has intuitive high energy efficiency[70]. In order to process larger graphs, the hosts may be involved and result in higher power consumption[94]. In Fig.4(a), accelerators with IDs by 1[16] and 2[29] can handle large graphs with good energy efficiency, which are both ASIC-based accelerators. This is because of the dedicated circuit designs and memory subsystems.

As for performance analysis, in spite that the results vary in different accelerators, the results show that the performance acts in a descend trend with graph size increasing. This is obvious for the BFS algorithm. Note that for the SSSP and PageRank algorithms, there is a lack of explicit evaluation results in existing literatures and only limited data points are depicted in Fig.4(b). Most of the results with high performance are based on a small graph size that the graph can fit into the on-chip/board memories. However, with graph size increasing, performance based on single accelerator decreases because external storages are often required[26,94]. Some designs based on multiple accelerators can maintain high performance when dealing with large graphs[28,86] because the graphs can still be held in on-chip/board memory.

Remarks. It gets clear that comparing different graph accelerators is extremely challenging due to the distinct evaluation parameters. To resolve this problem, the common practice in prior work is to compare the accelerator with some known systems as shown in Table 4. However, the compared systems used in different accelerators are still not comparable. For example, different accelerators adopt various strategies in preprocessing, parallel graph computation models, and runtime scheduling schemes. As a result, the accelerator evaluation and the peer comparison are still trapped into a deadlock. We conjecture that the lack of graph accelerator benchmarks and reference designs is the root of this problem. To this point, developing an open-sourced benchmark as well as an easy-to-port reference design can be a potential solution to make a fair evaluation.

6.3 Case Study: AccuGraph[15]

As a representative state-of-the-art FPGA-based graph processing accelerator, AccuGraph[15] has achieved impressive performance results with the dedicated hardware design for parallelizing the vertex updates that involve conflicts. For better understanding this survey, Fig.5 re-decomposes the original workflow of AccuGraph as a case study according to different stages that we have identified previously.

Preprocessing. For saving the space of on-chip memories, AccuGraph follows to use the compact graph representation with CSR. In an effort to balance the number of vertex accesses, AccuGraph presents an index-aware ordering to reorder the edges of each vertex by following a simple hash function of MOD(n) where n is up to the number of on-chip subgraph partitions. As for graph partition, considering that AccuGraph uses a pull-based model for high-throughput pipeline design, a vertex-cut graph partitioning method is used to ensure the sequential access of the in-degree edges of each vertex.

Parallel Graph Computation. AccuGraph is built upon a Xilinx Virtex Ultrascale+FPGA board. In order to avoid the half-bandwidth wasting problem of edge-centric programming model that simultaneously accesses both source and destination vertices, AccuGraph uses the vertex-centric programming model to access source vertices only for ensuring the sequential access of edges.

The core design of AccuGraph lies in a parallel accumulator with dedicated hardware circuits that can support the simultaneous update of conflicting vertices. The key insight is that the atomic operations of many graph algorithms manifest incremental and simplex features, which enables to execute massive conflicting vertex updates in an accumulative fashion. By handling atomic operations simultaneously and merging their results in parallel, the update operations for the same vertex can be therefore parallelized while preserving the correctness of final results.

It is also observed that a significant amount of locality exists for accessing associated edges of a particular active vertex. In order to further reduce the synchronization overhead of high-degree vertices, AccuGraph follows to use Copy-on-Write philosophy[124] to

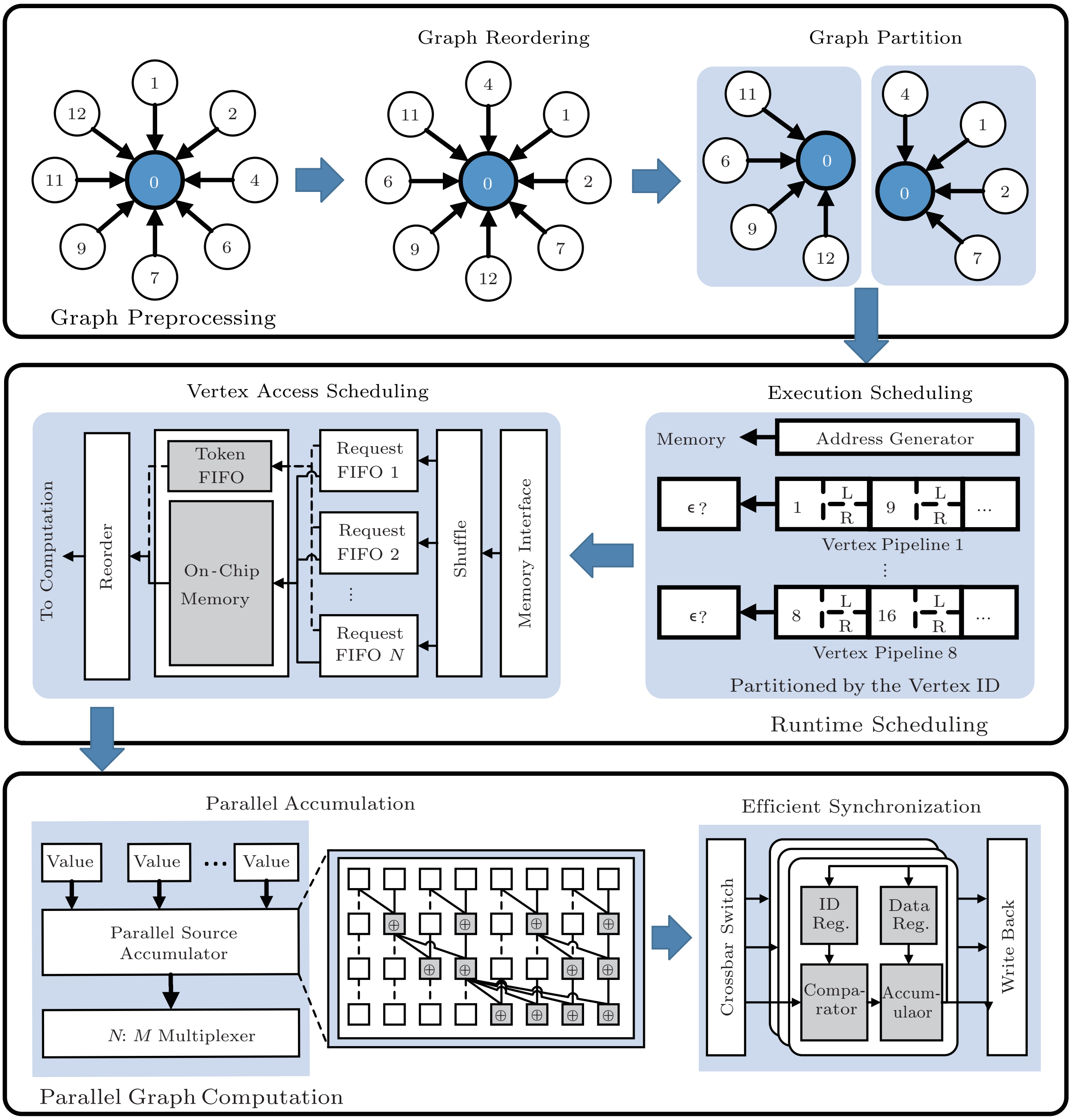


Fig.5. Workflow decomposition of AccuGraph in accordance with three major components (described in Fig.1) of preprocessing, parallel graph computation and runtime scheduling.

delay the writeback of vertex data. All intermediately-updated vertex data is stored into a specially designed scratchpad memory. If and only if all associated edges are finished, the updated value of a given vertex can be written into the main memory.

Runtime Scheduling. To better leverage the limited number of pins of parallel accumulator, AccuGraph uses an improved frontier-based scheduling. In the aspect of computational scheduling, it separately handles the pipelines of vertices and edges for reducing the out-of-order memory accesses. The edge pipelines access each edge sequentially while each edge pipeline dynamically adjusts the number of vertices to be processed via a degree-aware scheduling mechanism. As for memory access scheduling, the sparsity of graph often leads to the imbalance of accessing vertices. AccuGraph significantly enhances the throughput of on-chip computation by presenting an out-of-order approach for accessing the value of the vertex.

7 Challenges and Opportunities

With the recent efforts, graph processing accelerators have experienced a series of significant technical advances for achieving high throughput and energy efficiency. Nevertheless, there still has a long way for graph accelerators in practical use for many challenges. As emerging architectural technologies arise, we would also have great opportunities to make significant progresses in not only performance and energy efficiency but also supporting technologies for easy use, evaluation and maintenance.

7.1 Challenges

Programmability. The development and the execution of graph algorithms on existing accelerators rely deeply on the low-level programming with hardware description languages. This enforces that developers have to know the underlying hardware details. Programming for graph programs is non-trivial with a long development cycle. Though high-level programming languages, e.g., C/C++, make this relatively easy, there still lack the effective transformation and mapping of the high-level programming languages to the low-level hardware description languages. The general-purpose high-level synthesis (HLS) offers a viable solution, which is, however, potentially inefficient due to non-full consideration of graph characteristics. It is of great importance to build easy-to-use programming environments for graph processing accelerators.

Supporting Large Graphs. The scale of the graph size is still exploding, which can be easily beyond the available capacity of on-chip memories of a single graph accelerator. For supporting large graphs, an intuitive method is to extend to use larger memory for storing the whole graph. For example, we can use a cluster network of HMCs. However, this may cost a high price at routing the requisite data. An alternative approach is to use the heterogeneous graph processing. By using the host memory with more than Terabyte capacity, we can thus have sufficient memory space to store large graphs[26,28]. Also, a similar design is to connect multiple graph accelerators together and manage them uniformly[28,29]. Nevertheless, the problem is that a significant amount of communication overhead may occur between different graph accelerators.

Time-Evolving Graphs. Existing studies are mostly limited to static graph structures. The graph data may easily change in structure over time. Dynamic graph processing is a hot research topic[125−127]. For example, users of Twitter may update and delete a post at any time. They can also add and delete comments on this post. The complex and changeable graph data structure has a high requirement for the latency of graph accelerators. Some methods based on the incremental variation of the subgraph have achieved relatively good results under small-scale increments[126], but the efficient processing of the large-scale time-evolving graph is still an open problem.

Complex Attributes of Graphs. Different areas have different requirements for the attributes of graphs. For example, two nodes may involve a large number of associated edges that can be handled in parallel. This is common for the server links and road connections. In addition, a number of values can be also associated to a vertex or edge[128]. More complex is that the attributes of a graph in the graph network (GN) can be a vector, a set or even another graph[129]. These complex attributes of the graphs can result in totally different computing and memory requirements that existing graph processing research can neither fit nor be handled efficiently, let alone hardware circuit designs.

Machine Learning on Graphs. Deep learning or machine learning algorithms are also emerging on graphs. There are some research advances on how to represent graph structures into matrics[130,131]. This gives a new dimension of two emerging fields: machine learning and graph processing.

Hardware Interfaces. Almost all of existing graph processing accelerators are used solely. They work under the premise that the graph data is placed in its on-chip memory. For supporting large graphs as described previously, requisite external connections to either another accelerators or host processor are needed. This hence requires some extra interfaces for the connection and extension. Unfortunately, few customized graph processing accelerators have such kind of effective interfaces (instead of slow PCI Express lane connection) to support better communication and energy efficiency for graph processing.

Tool Chains. So far, there have been also no convenient tools for programmers to develop and use these graph accelerators easily. Particularly, if the graph programs come across the concurrency and performance bugs, programmers have to rebuild and re-wire the hardware circuit, which is notoriously costly. There still lacks a chain of utility tools for helping understand, diagnose or even fix these low-level problems during development.

Compiler Support. Compiler supporting is an effective way to fill the gap between high-level programming and low-level graph iteration. Symbolic execution is used to parallelize the dependent computations of vertices for achieving compelling performance results on general-purpose processors[132]. Execution parallelism can be also explored for irregular applications by aggressively scheduling execution dependencies at compile time[133]. However, more non-trivial efforts are still needed for graph processing accelerators to integrate these advanced compilation features due to the fact that existing (hardware and software) ecosystem surrounding graph accelerators are far from mature.

7.2 Opportunities

Widespread Adoption. To the best of our knowledge, graph processing has been used in many fields, e.g., social network, literature network, traffic network, and knowledge atlas. The earlier work focuses on addressing typical problems regarding graph searching, random walking, and graph clustering. Although there emerge a few latest advances that are attempting to solve the large, complex problems by leveraging graph processing[134], the application of graph processing still needs to expand. It is a series of open questions regarding how to leverage graph processing and further renovate its hardware acceleration to solve wider practical problems.

Emerging Technologies. As discussed before, a few recent studies have used emerging memory technologies (e.g., HMC and ReRAM) to accelerate graph processing, and made the good results in both performance and energy. Nevertheless, the potentials of these emerging technologies are still being under-utilized. For instance, GraphR[70] uses just one layer ReRAM only, but the fact is that the future ReRAM is often stacked. It is an interesting question on how to use the stacked ReRAM for graph processing acceleration in a more significant way in practice. To this point, more effective and efficient techniques for better supporting emerging technologies have to be settled.

FPGA on the Cloud. FPGAs have been widely adopted in industries to accelerate the datacenter[23] for the high energy efficiency and performance. FPGA providers such as Amazon, Baidu, and Tencent have also offered an easy and flexible programming environment for the FPGA development on the cloud. Users can directly program FPGA on the cloud with convenient GUI and sufficient open-source instances②. The abundant available FPGA resources and integrated development tools provide the opportunities for agile development of FPGA graph processing accelerators[22].

Rise of Specialized Architectures in Artificial Intelligence. There has emerged a number of AI specialized hardware accelerators in recent years[135,136]. These hardware accelerators have been used to accelerate machine learning applications in the cloud③. The abundant experience of existing AI accelerators can help us understand the underlying architecture design. Besides, a large number of educating resources and developing tools for AI accelerator development can promote the procedure of architecture designs. These opportunities brought by artificial intelligence accelerators can significantly improve the efficiency of graph processing accelerator development.

8 Conclusions

With the widely spreading graph applications, and gradually increasing data size and the complexity in big data analytics, the performance and the energy efficiency of graph processing have brought severe challenges to modern data processing eco-systems. There has emerged a large amount of work that aims at exploring software optimizations to improve the performance and energy efficiency of graph processing under general-purpose architectures, e.g., multi-core CPUs[52] and GPUs[6,8].

However, the significant gap between the unique feature of graph processing and the hardware features of general-purpose architectures limits the further improvement of performance and energy efficiency. Memory access efficiency suffers significantly from traditional memory hierarchy when facing the challenges of the intuitive features in graph processing, e.g., the irregularity and strong dependency[15,16]. GPUs also face the drawbacks, e.g., control and memory divergence, load imbalance and global memory access overhead[6].

②http://www.plunify.com/en/plunify-cloud/, Jan. 2019.

③http://cloud.google.com/tpu/, Jan. 2019.

That motivates the recent research efforts on developing new hardware architectures for graph processing.

With the trend and opportunities in domain-specific architectures[20], e.g., open-source implementations and agile chip development technics[22], customized graph processing accelerators have emerged as a promising solution to achieve both high performance and energy efficiency.

In this paper, we investigated a wide spectrum of studies on graph processing accelerators, and provided a systematic view on their design and implementation. Existing techniques have been categorized into three core aspects: preprocessing, parallel graph computation and runtime scheduling. For each aspect, we reviewed the state-of-the-art techniques and made our remarks on identifying the open problems for future research.

We also made a careful comparison of these studies, and highlighted the importance of evaluation benchmarks for graph processing accelerators. At last, we summarized the challenges and opportunities of graph processing accelerators, which, we believe, can help architect efficient graph processing accelerators. In summary, graph processing accelerators are still a hot research topic with many technical challenges and opportunities. We call for actions in this survey from different communities, including computer architectures, software systems, and databases, to respond these challenges cooperatively.

References

[1] Malewicz G, Austern M H, Bik A J, Dehnert J C, Horn I, Leiser N, Czajkowski G. Pregel: A system for large-scale graph processing. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, June 2010, pp.135-146.

[2] Low Y, Bickson D, Gonzalez J, Guestrin C, Kyrola A, Hellerstein J M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 2012, 5(8): 716-727.

[3] Shun J, Blelloch G E. Ligra: A lightweight graph processing framework for shared memory. In *Proc. the 18th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, February 2013, pp.135-146.

[4] Kyrola A, Blelloch G E, Guestrin C. GraphChi: Large-scale graph computation on just a PC. In *Proc. the 10th USENIX Conf. Operating Systems Design and Implementation*, October 2012, pp.31-46.

[5] Roy A, Mihailovic I, Zwaenepoel W. X-Stream: Edge-centric graph processing using streaming partitions. In *Proc. the 24th ACM SIGOPS Symp. Operating Systems Principles*, November 2013, pp.472-488.

[6] Zhong J, He B. Medusa: A parallel graph processing system on graphics processors. *ACM SIGMOD Record*, 2014, 43(2): 35-40.

[7] Khorasani F, Vora K, Gupta R, Bhuyan L N. CuSha: Vertex-centric graph processing on GPUs. In *Proc. the 23rd Int. Symp. High-Performance Parallel and Distributed Computing*, June 2014, pp.239-252.

[8] Wang Y, Davidson A, Pan Y, Wu Y, Riffel A, Owens J D. Gunrock: A high-performance graph processing library on the GPU. In *Proc. the 21st ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, March 2016, Article No. 11.

[9] Shi X, Luo X, Liang J, Zhao P, Di S, He B, Jin H. Frog: Asynchronous graph processing on GPU with hybrid coloring model. *IEEE Trans. Knowledge and Data Engineering*, 2018, 30(1): 29-42.

[10] Fu Z, Personick M, Thompson B. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proc. the 2nd International Workshop on Graph Data Management Experiences and Systems*, June 2014, Article No. 2.

[11] Liu H, Huang H H. Enterprise: Breadth-first graph traversal on GPUs. In *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis*, November 2015, Article No. 68.

[12] Beamer S, Asanovic K, Patterson D. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Proc. IEEE Int. Symp. Workload Characterization*, November 2015, pp.56-65.

[13] Malicevic J, Lepers B, Zwaenepoel W. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *Proc. the 2017 USENIX Annual Technical Conf.*, July 2017, pp.631-643.

[14] Nai L, Hadidi R, Sim J, Kim H, Kumar P, Kim H. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In *Proc. the 2007 IEEE Int. Symp. High Performance Computer Architecture*, February 2017, pp.457-468.

[15] Yao P, Zheng L, Liao X, Jin H, He B. An efficient graph accelerator with parallel data conflict management. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, November 2018, Article No. 8.

[16] Ham T J, Wu L, Sundaram N, Satish N, Martonosi M. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proc. the 49th Annual IEEE/ACM Int. Symp. Microarchitecture*, October 2016, Article No. 56.

[17] Nai L, Xia Y, Tanase I G, Kim H, Lin C Y. GraphBIG: Understanding graph computing in the context of industrial solutions. In *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis*, November 2015, Article No. 69.

[18] Satish N, Sundaram N, Patwary M M, Seo J, Park J, Hassaan M A, Sengupta S, Yin Z, Dubey P. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, June 2014, pp.979-990.

[19] Ben-Nun T, Sutton M, Pai S, Pingali K. Groute: An asynchronous multi-GPU programming model for irregular computations. In *Proc. the 22nd ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, February 2017, pp.235-248.

[20] Hennessy J, Patterson D. Domain specific architectures. In *Computer Architecture: A Quantitative Approach* (*6th edition*), Merken S, McFadden N (eds.), Elsevier, 2017, pp.540-606.

[21] Ceze L, Hill M D, Sankaralingam K, Wenisch T F. Democratizing design for future computing platforms. arXiv:1706.08597, 2017. http://arxiv.org/abs/1706.08597, Jun. 2017.

[22] Lee Y, Waterman A, Cook H *et al.* An agile approach to building RISC-V microprocessors. *IEEE Micro*, 2016, 36(2): 8-20.

[23] Caulfield A M, Chung E S, Putnam A *et al.* A cloud-scale acceleration architecture. In *Proc. the 49th Annual IEEE/ACM Int. Symp. Microarchitecture*, October 2016, Article No. 7.

[24] de Lorimier M, Kapre N, Mehta N *et al.* GraphStep: A system architecture for sparse-graph algorithms. In *Proc. the 14th Annual IEEE Symp. Field-Programmable Custom Computing Machines*, April 2006, pp.143-151.

[25] Attia O G, Johnson T, Townsend K, Jones P, Zambreno J. CyGraph: A reconfigurable architecture for parallel breadth-first search. In *Proc. the 2004 Int. Parallel and Distributed Processing Symp. Workshops*, May 2014, pp.228-235.

[26] Dai G, Chi Y, Wang Y, Yang H. FPGP: Graph processing framework on FPGA a case study of breadth-first search. In *Proc. the 2006 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, February 2016, pp.105-110.

[27] Zhou S, Chelmis C, Prasanna V K. High-throughput and energy-efficient graph processing on FPGA. In *Proc. the 24th IEEE Annual Int. Symp. Field-Programmable Custom Computing Machines*, May 2016, pp.103-110.

[28] Dai G, Huang T, Chi Y, Xu N, Wang Y, Yang H. ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture. In *Proc. the 2017 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, February 2017, pp.217-226.

[29] Ozdal M M, Yesil S, Kim T, Ayupov A, Greth J, Burns S, Özturk Ö. Energy efficient architecture for graph analytics accelerators. In *Proc. the 43rd ACM/IEEE Annual Int. Symp. Computer Architecture*, June 2016, pp.166-177.

[30] Zhou J, Liu S, Guo Q, Zhou X, Zhi T, Liu D, Wang C, Zhou X, Chen Y, Chen T. TuNao: A high-performance and energy-efficient reconfigurable accelerator for graph processing. In *Proc. the 17th IEEE/ACM Int. Symp. Cluster, Cloud and Grid Computing*, May 2017, pp.731-734.

[31] Ayupov A, Yesil S, Ozdal M M, Kim T, Burns S, Özturk Ö. A template-based design methodology for graph-parallel hardware accelerators. *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, 2018, 37(2): 420-430.

[32] Ahn J, Hong S, Yoo S, Mutlu O, Choi K. A scalable processing-in-memory accelerator for parallel graph processing. In *Proc. the 42nd ACM/IEEE Annual Int. Symp. Computer Architecture*, June 2015, pp.105-117.

[33] Pawlowski J T. Hybrid memory cube (HMC). In *Proc. the 23rd IEEE Hot Chips Symp.*, August 2011, Article No. 15.

[34] Kim J, Kim Y. HBM: Memory solution for bandwidth-hungry processors. In *Proc. the 26th IEEE Hot Chips Symp.*, August 2014, Article No. 19.

[35] Wong H S, Lee H Y, Yu S, Chen Y S, Wu Y, Chen P S, Lee B, Chen F T, Tsai M J. Metal-oxide RRAM. *Proceedings of the IEEE*, 2012, 100(6): 1951-1970.

[36] Page L, Brin S, Motwani R, Winograd T. The PageRank citation ranking: Bringing order to the web. Technical Report, Stanford InfoLab, 1999. http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf, Jan. 2019.

[37] McCune R R, Weninger T, Madey G. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Trans. Computing Surveys*, 2015, 48(2): Article No. 25.

[38] Shi X, Zheng Z, Zhou Y, Jin H, He L, Liu B, Hua Q. Graph processing on GPUs: A survey. *ACM Trans. Computing Surveys*, 2018, 50(6): Article No. 81.

[39] Heidari S, Simmhan Y, Calheiros R N, Buyya R. Scalable graph processing frameworks: A taxonomy and open challenges. *ACM Trans. Computing Surveys*, 2018, 51(3): Article No. 60.

[40] Gonzalez J E, Low Y, Gu H, Bickson D, Guestrin C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. the 10th USENIX Symp. Operating Systems Design and Implementation*, October 2012, pp.17-30.

[41] Avery C. Giraph: Large-scale graph processing infrastructure on Hadoop. In *Proc. the 2011 Hadoop Summit*, June 2011, pp.5-9.

[42] Gonzalez J E, Xin R S, Dave A, Crankshaw D, Franklin M J, Stoica I. GraphX: Graph processing in a distributed dataflow framework. In *Proc. the 11th USENIX Symp. Operating Systems Design and Implementation*, October 2014, pp.599-613.

[43] Teixeira C H, Fonseca A J, Serafini M, Siganos G, Zaki M J, Aboulnaga A. Arabesque: A system for distributed graph mining. In *Proc. the 25th Symp. Operating Systems Principles*, October 2015, pp.425-440.

[44] Chen R, Shi J, Chen Y, Chen H. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proc. the 10th European Conf. Computer Systems*, April 2015, Article No. 1.

[45] Zhu X, Chen W, Zheng W, Ma X. Gemini: A computation-centric distributed graph processing system. In *Proc. the 12th USENIX Symp. Operating Systems Design and Implementation*, November 2016, pp.301-316.

[46] Khayyat Z, Awara K, Alonazi A, Jamjoom H, Williams D, Kalnis P. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proc. the 8th ACM European Conf. Computer Systems*, April 2013, pp.169-182.

[47] Randles M, Lamb D, Taleb-Bendiab A. A comparative study into distributed load balancing algorithms for cloud computing. In *Proc. the 24th IEEE Int. Conf. Advanced Information Networking and Applications Workshops*, April 2010, pp.551-556.

[48] Zhao Y, Yoshigoe K, Xie M, Zhou S, Seker R, Bian J. LightGraph: Lighten communication in distributed graph-parallel processing. In *Proc. the 2004 IEEE Int. Congress on Big Data*, June 2014, pp.717-724.

[49] Wang P, Zhang K, Chen R, Chen H, Guan H. Replication-based fault-tolerance for large-scale graph processing. In *Proc. the 44th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks*, June 2014, pp.562-573.

[50] Nguyen D, Lenharth A, Pingali K. A lightweight infrastructure for graph analytics. In *Proc. the 24th ACM SIGOPS Symp. Operating Systems Principles*, November 2013, pp.456-471.

[51] Sundaram N, Satish N, Patwary M M, Dulloor S R, Anderson M J, Vadlamudi S G, Das D, Dubey P. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 2015, 8(11): 1214-1225.

[52] Zhang K, Chen R, Chen H. NUMA-aware graph-structured analytics. In *Proc. the 20th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, February 2015, pp.183-193.

[53] Han W S, Lee S, Park K, Lee J H, Kim M S, Kim J, Yu H. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proc. the 19th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, August 2013, pp.77-85.

[54] Yuan P, Zhang W, Xie C, Jin H, Liu L, Lee K. Fast iterative graph computation: A path centric approach. In *Proc. the 2004 Int. Conf. High Performance Computing, Networking, Storage and Analysis*, November 2014, pp.401-412.

[55] Zhu X, Han W, Chen W. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proc. the 2005 USENIX Annual Technical Conf.*, July 2015, pp.375-386.

[56] Chi Y, Dai G, Wang Y, Sun G, Li G, Yang H. NXgraph: An efficient graph processing system on a single machine. In *Proc. the 32nd IEEE Int. Conf. Data Engineering*, May 2016, pp.409-420.

[57] Maass S, Min C, Kashyap S, Kang W, Kumar M, Kim T. Mosaic: Processing a trillion-edge graph on a single machine. In *Proc. the 12th ACM European Conf. Computer Systems*, April 2017, pp.527-543.

[58] Seo H, Kim J, Kim M S. GStream: A graph streaming processing method for large-scale graphs on GPUs. In *Proc. the 20th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, February 2015, pp.253-254.

[59] Soman J, Kishore K, Narayanan P J. A fast GPU algorithm for graph connectivity. In *Proc. the 24th IEEE Int. Symp. Parallel & Distributed Processing, Workshops and PhD Forum*, April 2010, Article No. 87.

[60] McLaughlin A, Bader D A. Scalable and high performance betweenness centrality on the GPU. In *Proc. the 2014 Int. Conf. High Performance Computing, Networking, Storage and Analysis*, November 2014, pp.572-583.

[61] Sariyüce A E, Kaya K, Saule E, Çatalyürek Ü V. Betweenness centrality on GPUs and heterogeneous architectures. In *Proc. the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, March 2013, pp.76-85.

[62] Davidson A A, Baxter S, Garland M, Owens J D. Work-efficient parallel GPU methods for single-source shortest paths. In *Proc. the 28th IEEE Int. Parallel and Distributed Processing Symp.*, May 2014, pp.349-359.

[63] Hong S, Chafi H, Sedlar E, Olukotun K. Green-Marl: A DSL for easy and efficient graph analysis. In *Proc. the 17th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, March 2012, pp.349-362.

[64] Gharaibeh A, Reza T, Santos-Neto E, Costa L B, Sallinen S, Ripeanu M. Efficient large-scale graph processing on hybrid CPU and GPU systems. arXiv:1312.3018, 2013. http://arxiv.org/abs/1312.3018, Dec. 2018.

[65] Zhang T, Zhang J, Shu W, Wu M Y, Liang X. Efficient graph computation on hybrid CPU and GPU systems. *The Journal of Supercomputing*, 2015, 71(4): 1563-1586.

[66] Liu H, Huang H H, Hu Y. iBFS: Concurrent breadth-first search on GPUs. In *Proc. the 2016 Int. Conf. Management of Data*, June 2016, pp.403-416.

[67] Sengupta D, Song S L, Agarwal K, Schwan K. GraphReduce: Processing large-scale graphs on accelerator-based systems. In *Proc. the 2015 Int. Conf. High Performance Computing, Networking, Storage and Analysis*, November 2015, Article No. 28.

[68] Kim M S, An K, Park H, Seo H, Kim J. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proc. the 2016 Int. Conf. Management of Data*, June 2016, pp.447-461.

[69] Han L, Shen Z, Shao Z, Huang H H, Li T. A novel ReRAM-based processing-in-memory architecture for graph computing. In *Proc. the 6th IEEE Non-Volatile Memory Systems and Applications Symp.*, August 2017, Article No. 13.

[70] Song L, Zhuo Y, Qian X, Li H, Chen Y. GraphR: Accelerating graph processing using ReRAM. In *Proc. the 2018 IEEE Int. Symp. High Performance Computer Architecture*, February 2018, pp.531-543.

[71] Zhang J, Khoram S, Li J. Boosting the performance of FPGA-based graph processor using hybrid memory cube: A case for breadth first search. In *Proc. the 2017 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, February 2017, pp.207-216.

[72] Oguntebi T, Olukotun K. GraphOps: A dataflow library for graph analytics acceleration. In *Proc. the 2016 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, February 2016, pp.111-117.

[73] Dai G, Huang T, Chi Y, Zhao J, Sun G, Liu Y, Wang Y, Xie Y, Yang H. GraphH: A processing-in-memory architecture for large-scale graph processing. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems.* doi:10.1109/TCAD.2018.2821565.

[74] Zhang J, Li J. Degree-aware hybrid graph traversal on FPGA-HMC platform. In *Proc. the 2018 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, February 2018, pp.229-238.

[75] Zhou S, Kannan R, Min Y, Prasanna V K. FASTCF: FPGA-based accelerator for stochastic-gradient-descent-based collaborative filtering. In *Proc. the 2018 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, February 2018, pp.259-268.

[76] Khoram S, Zhang J, Strange M, Li J. Accelerating graph analytics by co-optimizing storage and access on an FPGA-HMC platform. In *Proc. the 2018 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, February 2018, pp.239-248.

[77] Han L, Shen Z, Liu D, Shao Z, Huang H H, Li T. A novel ReRAM-based processing-in-memory architecture for graph traversal. *ACM Trans. Storage*, 2018, 14(1): Article No. 9.

[78] Wang Q, Jiang W, Xia Y, Prasanna V. A message-passing multi-softcore architecture on FPGA for breadth-first search. In *Proc. the 2010 Int. Conf. Field-Programmable Technology*, December 2010, pp.70-77.

[79] Umuroglu Y, Morrison D, Jahre M. Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform. In *Proc. the 25th Int. Conf. Field Programmable Logic and Applications*, September 2015, Article No. 12.

[80] Zhou S, Prasanna V K. Accelerating graph analytics on CPU-FPGA heterogeneous platform. In *Proc. the 29th Int. Symp. Computer Architecture and High Performance Computing*, October 2017, pp.137-144.

[81] Zhang M, Zhuo Y, Wang C, Gao M, Wu Y, Chen K, Kozyrakis C, Qian X. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *Proc. the 2018 IEEE Int. Symp. High Performance Computer Architecture*, February 2018, pp.544-557.

[82] Huang T, Dai G, Wang Y, Yang H. HyVE: Hybrid vertex-edge memory hierarchy for energy-efficient graph processing. In *Proc. the 2018 Design, Automation and Test in Europe Conference and Exhibition*, March 2018, pp.973-978.

[83] Ozdal M M, Yesil S, Kim T, Ayupov A, Greth J, Burns S, Ozturk O. Graph analytics accelerators for cognitive systems. *IEEE Micro*, 2017, 37(1): 42-51.

[84] Kapre N. Custom FPGA-based soft-processors for sparse graph acceleration. In *Proc. the 26th IEEE Int. Conf. Application-Specific Systems, Architectures and Processors*, July 2015, pp.9-16.

[85] Betkaoui B, Thomas D B, Luk W, Przulj N. A framework for FPGA acceleration of large graph problems: Graphlet counting case study. In *Proc. the 2011 Int. Conf. Field-Programmable Technology*, December 2011, Article No. 2.

[86] Betkaoui B, Wang Y, Thomas D B, Luk W. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *Proc. the 23rd IEEE Int. Conf. Application-Specific Systems, Architectures and Processors*, July 2012, pp.8-15.

[87] Betkaoui B, Wang Y, Thomas D B, Luk W. Parallel FPGA-based all pairs shortest paths for sparse networks: A human brain connectome case study. In *Proc. the 22nd Int. Conf. Field Programmable Logic and Applications*, August 2012, pp.99-104.

[88] Nurvitadhi E, Weisz G, Wang Y, Hurkat S, Nguyen M, Hoe J C, Martínez J F, Guestrin C. GraphGen: An FPGA framework for vertex-centric graph computation. In *Proc. the 22nd IEEE Annual Int. Symp. Field-Programmable Custom Computing Machines*, May 2014, pp.25-28.

[89] Attia O G, Grieve A, Townsend K R, Jones P, Zambreno J. Accelerating all-pairs shortest path using a message-passing reconfigurable architecture. In *Proc. the 2015 Int. Conf. Reconfigurable Computing and FPGAs*, December 2015, Article No. 5.

[90] Engelhardt N, So H K. GraVF: A vertex-centric distributed graph processing framework on FPGAs. In *Proc. the 26th Int. Conf. Field Programmable Logic and Applications*, August 2016, Article No. 62.

[91] Jin H, Yao P, Liao X, Zheng L, Li X. Towards dataflow-based graph accelerator. In *Proc. the 37th IEEE Int. Conf. Distributed Computing Systems*, June 2017, pp.1981-1992.

[92] Zhou S, Chelmis C, Prasanna V K. Accelerating large-scale single-source shortest path on FPGA. In *Proc. the 2015 Int. Parallel and Distributed Processing Symposium Workshop*, May 2015, pp.129-136.

[93] Zhou S, Chelmis C, Prasanna V K. Optimizing memory performance for FPGA implementation of PageRank. In *Proc. the 2015 Int. Conf. Reconfigurable Computing and FPGAs*, December 2015, Article No. 53.

[94] Jun S W, Wright A, Zhang S, Xu S, Arvind. GraFBoost: Using accelerated flash storage for external graph analytics. In *Proc. the 45th ACM/IEEE Int. Symp. Computer Architecture*, June 2018, pp.411-424.

[95] Thomas D, Moorby P. The Verilog® Hardware Description Language (5th edition). Springer, 2002.

[96] Ashenden P J. The Designer's Guide to VHDL (3rd edition). Morgan Kaufmann, 2008.

[97] Lee J, Kim H, Yoo S, Choi K, Hofstee H P, Nam G J, Nutter M R, Jamsek D. ExtraV: Boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment*, 2017, 10(12): 1706-1717.

[98] Kim G, Kim J, Ahn J H, Kim J. Memory-centric system interconnect design with hybrid memory cubes. In *Proc. the 22nd Int. Conf. Parallel Architectures and Compilation Techniques*, September 2013, pp.145-155.

[99] Xu C, Niu D, Muralimanohar N, Balasubramonian R, Zhang T, Yu S, Xie Y. Overcoming the challenges of crossbar resistive memory architectures. In *Proc. the 21st IEEE Int. Symp. High Performance Computer Architecture*, February 2015, pp.476-488.

[100] Do J, Kee Y S, Patel J M, Park C, Park K, DeWitt D J. Query processing on smart SSDs: Opportunities and challenges. In *Proc. the 2013 ACM SIGMOD Int. Conf. Management of Data*, June 2013, pp.1221-1230.

[101] Jun S W, Liu M, Lee S, Hicks J, Ankcorn J, King M, Xu S, Arvind. BlueDBM: An appliance for big data analytics. In *Proc. the 42nd ACM Annual Int. Symp. Computer Architecture*, June 2015, pp.1-13.

[102] Zhang J, Jung M. FlashAbacus: A self-governing flash-based accelerator for low-power systems. In *Proc. the 13th EuroSys Conf.*, April 2018, Article No. 15.

[103] Ozdal M M. Emerging accelerator platforms for data centers. *IEEE Design & Test*, 2018, 35(1): 47-54.

[104] Weisz G, Melber J, Wang Y, Fleming K, Nurvitadhi E, Hoe J C. A study of pointer-chasing performance on shared-memory processor-FPGA systems. In *Proc. the 2016 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, February 2016, pp.264-273.

[105] Gu B, Yoon A S, Bae D H, Jo I, Lee J, Yoon J, Kang J U, Kwon M, Yoon C, Cho S, Jeong J, Chang D. Biscuit: A framework for near-data processing of big data workloads. In *Proc. the 43rd Int. Symp. Computer Architecture*, June 2016, pp.153-165.

[106] Son Y, Choi J, Jeon J, Min C, Kim S, Yeom H Y, Han H. SSD-assisted backup and recovery for database systems. In *Proc. the 33rd IEEE Int. Conf. Data Engineering*, April 2017, pp.285-296.

[107] Song W S, Gleyzer V, Lomakin A, Kepner J. Novel graph processor architecture, prototype system, and results. In *Proc. the 2016 IEEE High Performance Extreme Computing Conference*, September 2016, Article No. 59.

[108] Jin H, Yao P, Liao X. Towards dataflow based graph processing. *Science China Information Sciences*, 2017, 60(12): Article No. 126102.

[109] Windh S, Budhkar P, Najjar W A. CAMs as synchronizing caches for multithreaded irregular applications on FPGAs. In *Proc. the 2015 ACM/IEEE Int. Conf. Computer-Aided Design*, November 2015, pp.331-336.

[110] Wang L, Yang X, Dai H. Scratchpad memory allocation for arrays in permutation graphs. *Science China Information Sciences*, 2013, 56(5): 1-13.

[111] Gao M, Ayers G, Kozyrakis C. Practical near-data processing for in-memory analytics frameworks. In *Proc. the 2015 Int. Conf. Parallel Architecture and Compilation*, October 2015, pp.113-124.

[112] Faloutsos M, Faloutsos P, Faloutsos C. On power-law relationships of the Internet topology. *ACM SIGCOMM Computer Communication Review*, 1999, 29(4): 251-262.

[113] Xie C, Chen R, Guan H, Zang B, Chen H. SYNC or ASYNC: Time to fuse for distributed graph-parallel computation. In *Proc. the 20th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, February 2015, pp.194-204.

[114] Ozdal M M, Yesil S, Kim T, Ayupov A, Burns S, Ozturk O. Architectural requirements for energy efficient execution of graph analytics applications. In *Proc. the 2015 IEEE/ACM Int. Conf. Computer-Aided Design*, November 2015, pp.676-681.

[115] Beamer S, Asanović K, Patterson D. Direction-optimizing breadth-first search. In *Proc. the 2012 Int. Conf. High Performance Computing, Networking, Storage and Analysis*, November 2012, Article No. 12.

[116] Beamer S, Asanović K, Patterson D. The GAP benchmark suite. arXiv:1508.03619, 2015. http://arxiv.org/abs/1508.03619, May 2017.

[117] Scarpazza D P, Villa O, Petrini F. Efficient breadth-first search on the Cell/B.E. processor. *IEEE Trans. Parallel and Distributed Systems*, 2008, 19(10): 1381-95.

[118] Milenković T, Lai J, Pržulj N. GraphCrunch: A tool for large network analyses. *BMC Bioinformatics*, 2008, 9: Article No. 70.

[119] Hong S, Oguntebi T, Olukotun K. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proc. the 2011 Int. Conf. Parallel Architectures and Compilation Techniques*, October 2011, pp.78-88.

[120] Matsumoto K, Nakasato N, Sedukhin S G. Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system. In *Proc. the 13th IEEE Int. Conf. High Performance Computing and Communications*, September 2011, pp.145-152.

[121] Siek J G, Lee L Q, Lumsdaine A. The Boost Graph Library: User Guide and Reference Manual (PAP/CDR edition). Addison-Wesley Professional, 2001.

[122] Ma X, Zhang D, Chiou D. FPGA-accelerated transactional execution of graph workloads. In *Proc. the 2017 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, February 2017, pp.227-236.

[123] Zheng D, Mhembere D, Burns R, Vogelstein J, Priebe C E, SzalayA S. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *Proc. the 13th USENIX Conf. File and Storage Technologies*, February 2015, pp.45-58.

[124] Rodeh O. B-trees, shadowing, and clones. *ACM Transactions on Storage*, 2008, 3(4): Article No. 2.

[125] Sha M, Li Y, He B, Tan K L. Accelerating dynamic graph analytics on GPUs. *Proceedings of the VLDB Endowment*, 2017, 11(1): 107-120.

[126] Shi X, Cui B, Shao Y, Tong Y. Tornado: A system for real-time iterative analysis over evolving data. In *Proc. the 2016 Int. Conf. Management of Data*, June 2016, pp.417-430.

[127] Chen H, Sun Z, Yi F, Su J. BufferBank storage: An economic, scalable and universally usable in-network storage model for streaming data applications. *Science China Information Sciences*, 2016, 59(1): 1-15.

[128] Zhang M, Wu Y, Chen K, Qian X, Li X, Zheng W. Exploring the hidden dimension in graph processing. In *Proc. the 12th USENIX Conf. Operating Systems Design and Implementation*, November 2016, pp.285-300.

[129] Battaglia P W, Hamrick J B, Bapst V *et al.* Relational inductive biases, deep learning, and graph networks. arXiv:1806.01261, 2018. http://arxiv.org/abs/1806.01261, Jun. 2018.

[130] Narayanan A, Chandramohan M, Venkatesan R, Chen L, Liu Y, Jaiswal S. graph2vec: Learning distributed representations of graphs. arXiv:1707.05005, 2017. https://arxiv.org/abs/1707.05005, Jun. 2018.

[131] Ribeiro L F, Saverese P H, Figueiredo D R. Struc2vec: Learning node representations from structural identity. In *Proc. the 23rd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, August 2017, pp.385-394.

[132] Zheng L, Liao X, Jin H. Efficient and scalable graph parallel processing with symbolic execution. *ACM Trans. Architecture and Code Optimization*, 2018, 15(1): Article No. 3.

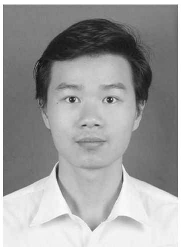
[133] Li Z, Liu L, Deng Y, Yin S, Wang Y, Wei S. Aggressive pipelining of irregular applications on reconfigurable hardware. In *Proc. the 44th Annual Int. Symp. Computer Architecture*, June 2017, pp.575-586.

[134] Zheng L, Liao X, Jin H, Zhao J, Wang Q. Scalable concurrency debugging with distributed graph processing. In *Proc. the 2018 Int. Symp. Code Generation and Optimization*, February 2018, pp.188-199.

[135] Jouppi N P, Young C, Patil N *et al.* In-datacenter performance analysis of a tensor processing unit. In *Proc. the 44th Annual Int. Symp. Computer Architecture*, June 2017, pp.1-12.

[136] Chen T, Du Z, Sun N, Wang J, Wu C, Chen Y, Temam O. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proc. the 19th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, March 2014, pp.269-284.

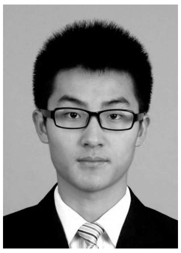
Chuang-Yi Gui is currently a Ph.D. candidate in the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), Wuhan. He received his B.E. degree in information security at HUST, Wuhan, in 2017. His current research interests include graph processing and reconfigurable computing.



Long Zheng is now a postdoctoral researcher in the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), Wuhan. He received his Ph.D. degree in computer engineering at HUST, Wuhan, in 2016. His current research interests include program analysis, runtime systems, and configurable computer architecture with a particular focus on graph processing.

Bingsheng He is currently an associate professor at Department of Computer Science, School of Computing, National University of Singapore (NUS), Singapore. Before joining NUS in May 2016, he held a research position in the System Research group of Microsoft Research Asia (2008-2010) and a faculty position in Nanyang Technological University, Singapore. He got his Bachelor's degree in computer science and engineering in Shanghai Jiao Tong University (1999-2003), Shanghai, and his Ph.D. degree in computer science in Hong Kong University of Science & Technology (2003-2008), Hong Kong. His current research interests include big data management systems (with special interests in cloud computing and emerging hardware systems), parallel and distributed systems and cloud computing.

Cheng Liu is an associate professor of Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. He received his B.E. and M.E. degree in microelectronic engineering from Harbin Institute of Technology, Harbin, in 2009 and his Ph.D. degree in computer engineering from The University of Hong Kong, Hong Kong, in 2016. His research focuses on FPGA-based reconfigurable computing and domain-specific computing.



Xin-Yu Chen is now a Ph.D. candidate of computer science in the National University of Singapore, Singapore. He received his B.E. degree in electronic science and technology from Harbin Institute of Technology, Weihai, in 2016. His current research interests include FPGA-based heterogeneous computing and database systems.

Xiao-Fei Liao received his Ph.D. degree in computer science and engineering from Huazhong University of Science and Technology (HUST), Wuhan, in 2005. He is now the vice dean in the School of Computer Science and Technology at HUST, Wuhan. He has served as a reviewer for many conferences and journal papers. His research interests are in the areas of system software, P2P system, cluster computing and streaming services. He is a member of IEEE and the IEEE Computer Society.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST), Wuhan. Jin received his Ph.D. degree in computer engineering from HUST, Wuhan, in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. Jin is a fellow of CCF and IEEE, and a member of ACM. He has co-authored 15 books and published over 600 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.