JCST Journal of Computer Science & Technology

Indexed in: SCIE, EI, INSPEC, CBST, DBLP, etc.

Sponsored by: ICT, CAS & CCF

Towards High-Performance Graph Processing: From a Hardware/Software Co-Design Perspective

Liao Xiao-Fei, Zhao Wen-Ju, Jin Hai, Yao Peng-Cheng, Huang Yu, Wang Qing-Gang, Zhao Jin, Zheng Long, Zhang Yu, Shao Zhi-Yuan

View online: http://doi.org/10.1007/s11390-024-4150-0

Articles you may be interested in

A Survey on Graph Processing Accelerators: Challenges and Opportunities

Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, Hai Jin

Journal of Computer Science and Technology. 2019, 34(2): 339-371 http://doi.org/10.1007/s11390-019-1914-z

Ad Hoc File Systems for High-Performance Computing

André Brinkmann, Kathryn Mohror, Weikuan Yu, Philip Carns, Toni Cortes, Scott A. Klasky, Alberto Miranda, Franz-Josef Pfreundt, Robert B. Ross, Marc-André Vef

Journal of Computer Science and Technology. 2020, 35(1): 4-26 http://doi.org/10.1007/s11390-020-9801-1

Interference Analysis of Co-Located Container Workloads: A Perspective from Hardware Performance Counters

Wen-Yan Chen, Ke-Jiang Ye, Cheng-Zhi Lu, Dong-Dai Zhou, Cheng-Zhong Xu

Journal of Computer Science and Technology. 2020, 35(2): 412-417 http://doi.org/10.1007/s11390-020-9707-y

FDGLib: A Communication Library for Efficient Large-Scale Graph Processing in FPGA-Accelerated Data Centers

Yu-Wei Wu, Qing-Gang Wang, Long Zheng, Xiao-Fei Liao, Hai Jin, Wen-Bin Jiang, Ran Zheng, Kan Hu

Journal of Computer Science and Technology. 2021, 36(5): 1051-1070 http://doi.org/10.1007/s11390-021-1242-y

Mochi: Composing Data Services for High-Performance Computing Environments

Robert B. Ross, George Amvrosiadis, Philip Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K. Gutierrez, Robert Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, Qing Zheng

Journal of Computer Science and Technology. 2020, 35(1): 121-144 http://doi.org/10.1007/s11390-020-9802-0

Developer Role Evolution in Open Source Software Ecosystem: An Explanatory Study on GNOME

Can Cheng, Bing Li, Zeng-Yang Li, Yu-Qi Zhao, Feng-Ling Liao

Journal of Computer Science and Technology. 2017, 32(2): 396-414 http://doi.org/10.1007/s11390-017-1728-9

JCST Homepage: https://jcst.ict.ac.cn
SPRINGER Homepage: https://www.springer.com/journal/11390
E-mail: jcst@ict.ac.cn
Online Submission: https://mc03.manuscriptcentral.com/jcst

JCST Official
WeChat Account

JCST WeChat
Service Account

Twitter: JCST_Journal
LinkedIn: Journal of Computer Science and Technology

Liao XF, Zhao WJ, Jin H *et al.* Towards high-performance graph processing: From a hardware/software co-design perspective. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 39(2): 245-266 Mar. 2024. DOI: 10.1007/s11390-024-4150-0

Towards High-Performance Graph Processing: From a Hardware/Software Co-Design Perspective

Xiao-Fei Liao[1, 2, 3] (廖小飞), *Distinguished Member, CCF, Member, IEEE*
Wen-Ju Zhao[1, 2, 3] (赵文举), *Student Member, CCF*
Hai Jin[1, 2, 3, *] (金　海), *Fellow, CCF, IEEE, Life Member, ACM*
Peng-Cheng Yao[1, 2, 3, 4] (姚鹏程), *Member, CCF*, Yu Huang[1, 2, 3, 4] (黄　禹), *Member, CCF*
Qing-Gang Wang[1, 2, 3, 4] (王庆刚), *Member, CCF*, Jin Zhao[1, 2, 3, 4] (赵　进), *Member, CCF, ACM, IEEE*
Long Zheng[1, 2, 3, 4] (郑　龙), *Senior Member, CCF, Member, ACM, IEEE*
Yu Zhang[1, 2, 3, 4] (张　宇), *Senior Member, CCF, Member, ACM, IEEE*
and Zhi-Yuan Shao[1, 2, 3, 4] (邵志远), *Member, CCF*

[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] *Zhejiang Lab, Hangzhou 311121, China*

E-mail: xfliao@hust.edu.cn; wjzh@hust.edu.cn; hjin@hust.edu.cn; pcyao@hust.edu.cn; yuh@hust.edu.cn; qgwang@hust.edu.cn zjin@hust.edu.cn; longzh@hust.edu.cn; zhyu@hust.edu.cn; zyshao@hust.edu.cn

Received January 26, 2024; accepted March 3, 2024.

Abstract Graph processing has been widely used in many scenarios, from scientific computing to artificial intelligence. Graph processing exhibits irregular computational parallelism and random memory accesses, unlike traditional workloads. Therefore, running graph processing workloads on conventional architectures (e.g., CPUs and GPUs) often shows a significantly low compute-memory ratio with few performance benefits, which can be, in many cases, even slower than a specialized single-thread graph algorithm. While domain-specific hardware designs are essential for graph processing, it is still challenging to transform the hardware capability to performance boost without coupled software codesigns. This article presents a graph processing ecosystem from hardware to software. We start by introducing a series of hardware accelerators as the foundation of this ecosystem. Subsequently, the codesigned parallel graph systems and their distributed techniques are presented to support graph applications. Finally, we introduce our efforts on novel graph applications and hardware architectures. Extensive results show that various graph applications can be efficiently accelerated in this graph processing ecosystem.

Keywords graph processing, hardware accelerator, software system, high performance, ecosystem

1 Introduction

Graphs are potent data structures capable of modeling complex relationships. They have been used in various domains, including but not limited to recommendation systems, brain analysis, energy management, and cybersecurity[1–6]. To fully exploit the rich information inherent in graphs, many researchers are dedicating their efforts to graph processing, which is gaining popularity in the community.

Cover Article
This work was supported by the National Key Research and Development Program of China under Grant No. 2023YFB-4502300.
*Corresponding Author

However, efficient processing of large graphs is challenging. In particular, graphs have few connections between nodes, and the number of edges connected to a node follows a power-law distribution, resulting in extreme sparsity and irregularity. These characteristics lead to challenges such as computation conflict, random memory access, and irregular communication when processing graph workloads. These challenges significantly diminish the efficiency of graph processing[7–16].

We propose an integrated graph processing ecosystem to achieve the full potential of graph processing. As shown in Fig.1, this ecosystem provides comprehensive solutions for graph processing, from hardware design to software system. These solutions are designed to achieve a high throughput, low latency, low power, and scalable graph processing architecture.

In terms of hardware accelerators design, we propose a family of graph hardware accelerators based on different memory architectures. As shown in Table 1, we first introduce AccuGraph[9], a pioneering solution that addresses the graph data conflict problem based on DRAM. Furthermore, we explore graph accelerators based on high-bandwidth memory (HBM) and design ScalaBFS2[17] and ScalaGraph[13], which aim to improve the efficiency of HBM and the scalability of processing elements (PEs), respectively.

In terms of graph software systems, our focus lies in optimizing the efficiency of graph processing so that they fully utilize different hardware resources. As shown in Table 1, our software libraries include HotGraph[18]/GraphFly[19]/CGraph[20], DiGraph[21], GraSu[22], and FBSGraph[23], corresponding to CPU, GPU, FPGA, and distributed platforms, respectively. They are dedicated to enhancing the parallelism and locality of graph processing systems by carefully designing data structures and scheduling strategies.

Furthermore, recent years have witnessed a surge in novel graph processing workloads and architectures, which pose new challenges to existing graph processing architectures. To tackle emerging workloads, including graph construction, hypergraphs, and heterogeneous graphs, we propose hardware-software co-design solutions, such as FNNG[24], XuLin[25], and MetaNMP[26]. To harness the potential of emerging architectures like Processing In/Near Memory, we propose innovative solutions based on ReRAM and CMOS, including Spara[27], ReFlip[28], and Hetraph[29]. As shown in Table 1, these solutions enhance data processing efficiency and parallelism, leading to a significant improvement in graph processing efficiency.

The remaining sections provide detailed introductions. In Section 2, we detail the design of hardware accelerators. In Section 3, we introduce graph software systems. In Section 4, we first discuss the processing of emerging graph workloads and then describe how we fully utilize the emerging hardware. Fi-

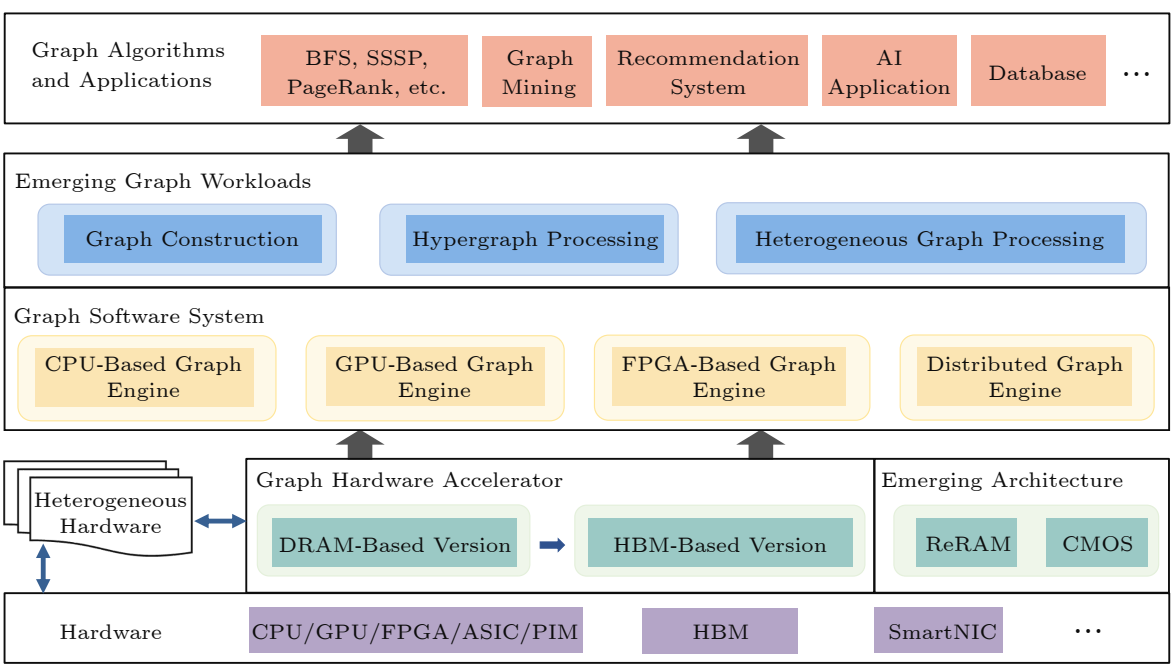


Fig.1. Graph processing ecosystem. BFS: Breath First Search; SSSP: Single Source Shortest Path.

Table 1. Graph Processing Ecosystem: From Hardware Accelerators, Software Systems to Emerging Graph Workloads and Architectures

Categorization		Name	Algorithm	Challenge	Speedup (Baseline)
Hardware accelerator	DRAM	AccuGraph[9]	BFS, SSSP, PageRank, etc.	How to design an efficient accumulator for parallelizing the conflicting data accesses for vertex updates	3.14x (ForeGraph)
	HBM	ScalaBFS2[17]	BFS	How to utilize HBM bandwidth efficiently in FPGAs	1.34x–2.40x (GPU-Gunrock)
		ScalaGraph[13]	BFS, SSSP, PageRank, etc.	How to realize a distributed on-chip memory in a graph accelerator with HBM	3.20x (GPU-Gunrock)
Software system	CPU	HotGraph[18] (static)	BFS, SSSP, PageRank, etc.	How to accelerate cross-partition status updates and convergence	5.22x (Maiter)
		GraphFly[19] (dynamic)	BFS, SSSP, PageRank, etc.	How to reuse the data accessed through refinement in recomputation	5.81x (KickStarter)
		CGraph[20] (concurrent)	BFS, SSSP, PageRank, etc.	How to leverage the correlations between CGP jobs to improve throughput	4.32x (Nxgraph)
	GPU	DiGraph[21]	BFS, SSSP, PageRank, etc.	How to make iterative directed graph processing converge faster and have lower data access costs	3.54x (Groute)
	FPGA	GraSu[22]	BFS, SSSP, PageRank, etc.	How to build dynamic graph accelerators based on existing static graph accelerators with minimal costs	34.24x (Stinger)
	Distributed	FBSGraph[23]	BFS, SSSP, PageRank, etc.	How to accelerate vertex state propagation with low overhead on distributed platforms	1.70x (Maiter)
Emerging graph workloads and architecture	Emerging graph workloads	FNNG[24]	NN-Descent	How to decrease the memory and computation overhead of NN-Desent from the architectural level	2.10x (GNND)
		XuLin[25]	PageRank, BC, CC, etc.	How to minimize off-chip communication traffic in hypergraph processing	8.77x (ChGraph)
		MetaNMP[26]	HGNN, GNN	How to reduce the large memory footprint and severe redundant computation in HGNN	415.18x (GPU-HGNN)
	Emerging architecture	Spara[27]	BFS, SSSP, PageRank, etc.	How to reduce inefficient computation in graph processing with ReRAM-based accelerators	8.21x (GraphR)
		ReFlip[28]	GCN	How to use crossbar-based PIM to unify GCN executions	15.63x (AWB-GCN)
		Hetraph[29]	BFS, SSSP, PageRank, etc.	How to efficiently use heterogeneous PIM to accelerate graph processing tasks	1.56x (GPU-Gunrock)

nally, in Section 5, we conclude our work with a summary.

2 Graph Hardware Accelerator

Several efforts have been dedicated to improving the performance of graph processing. However, because of the irregularity of graphs, general-purpose processors (e.g., CPUs/GPUs) suffer from severe computational conflicts, memory inefficiencies, and poor scalability when running graph applications. This section gives solutions to the above problems with different memories.

2.1 AccuGraph: A DRAM-Based Version

In the past few years, there has been substantial research on domain-specific accelerators to improve memory performance for irregular graph data[30, 31]. Despite these research efforts, graph applications still suffer from suboptimal performance due to irregular computation patterns. When traversing the graph topology, multiple vertices might access the same vertex simultaneously, resulting in substantial data conflicts.

Existing graph accelerators adopt atomic structures (e.g., content addressable memory[30]) to sequentialize the process of conflicting operations. While the atomic structures ensure the correctness of execution, they significantly degrade performance since only one conflicting operation can be processed at a time. From our experiment, existing graph accelerators stall in at most 40% of the time due to atomic protection.

Fortunately, conflicting operations in typical graph applications have two significant characteristics. First, the atomic operations performed on differ-

ent edges in the graph follow the commutative and associative law. Second, the atomic operations used to update the conflicting vertex's value are simple and executed repeatedly. These two observations introduce the opportunity to eliminate deficiency caused by the conflict updates inside the vertex.

Based on these observations, we present AccuGraph, a novel graph accelerator that processes conflicting operations in parallel. As shown in Fig.2(a), multiple conflicting operations are simultaneously scheduled to improve computation efficiency. A specialized accumulator is provided to avoid potential data hazards by merging all operations targeting the same vertex into one operation. Note that these merging operations are not always addition. For example, in SSSP, this accumulator performs a min operation, while in PageRank, it performs an addition operation. But we uniformly call it an accumulator. AccuGraph significantly reduces pipeline stalls and achieves nearly fully pipelined computation by parallelizing conflicting vertex updates by using this accumulator.

Nevertheless, designing this efficient accumulator to merge operations is difficult because of two challenges. First, multiple low-degree vertices are expected to process simultaneously for efficient parallelism. However, establishing a definite mapping rule from inputs to outputs proven to be tough because the degrees of vertices vary dynamically. Second, especially for high-degree vertices with numerous edges, processing multiple edges simultaneously becomes challenging due to the limited width of the accumulator.

For the first challenge, we find that the low-degree vertex accumulation is a variation of the prefix-sum problem. Let us assume N update values belonging to M vertices must be processed at the same time. This challenge can be described by $p_j = \sum_{1 \leqslant i \leqslant N} a_i \times b_{ij}$, $1 \leqslant j \leqslant M$, where p_j denotes the accumulated result of vertex j. a_i presents the update value i, and b_{ij} indicates whether a_i belongs to vertex j. In other words, the low-degree vertex accumulation needs to compute the prefix sum of multiple vertices at runtime.

According to the formalization, we advocate a novel parallel accumulator shown in Fig.2(b). This accumulator is designed by using the Ladner-Fischer accumulator, an existing well-established prefix-sum accumulator, as the foundation. The number of vertices to be scheduled is determined on-the-fly by the degree-aware scheduling. After that, the source vertex accumulator simultaneously accumulates updated values of different destination vertices. To preserve the correctness, we complement a breakpoint recognizing mechanism. Specifically, AccuGraph attaches each update with the ID of its destination vertex, which is leveraged as a signal to ensure that multiple updates are accumulated to the same destination.

After that, the $N \times M$ multiplexer delivers the accumulated results to each destination vertex sequentially. To address the second challenge, we advocate a destination vertex accumulator. It merges the accumulated values with a decision to delay the write-back of the corresponding destination vertex data. This mechanism eliminates synchronization overhead on the temporary vertex data and massive random

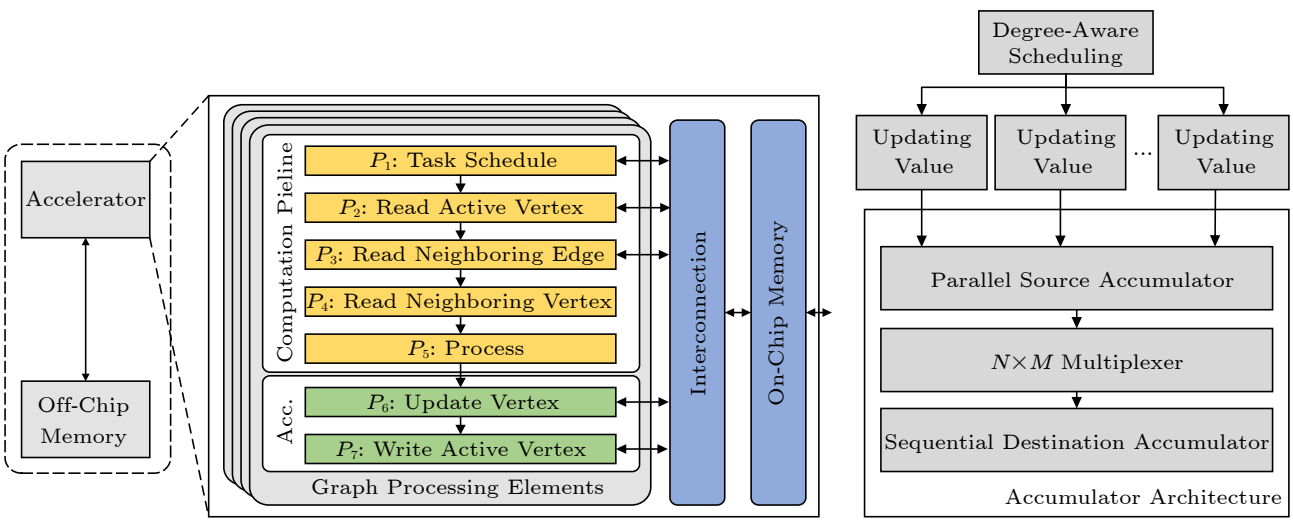


Fig.2. (a) Architecture of AccuGraph and (b) detailed design of the parallel accumulator[9]. P_i denotes the i-th pipeline stage. Acc. means accumulator.

edge accesses.

Fig.3 depicts the normalized performance of AccuGraph compared with ForeGraph[31]. From our experimental data, AccuGraph delivers an average of 2.36 GTEPS. This performance marks a notable advantage, as it outperforms ForeGraph, achieving a speedup of up to 3.14x.

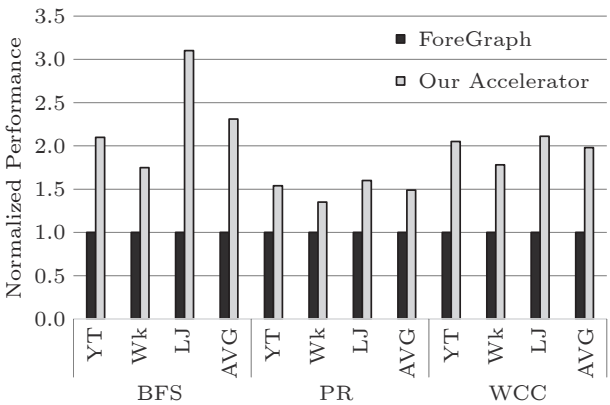


Fig.3. AccuGraph normalized to the ForeGraph performance[9]. YT: Youtube; Wk: Wiki; LJ: LiveJournal; AVG: average; PR: PageRank; WCC: Weakly Connected Components.

2.2 ScalaBFS2: An HBM-Based Version

With the latest advancements in 3D stacking technology, HBM is proposed to enhance memory bandwidth. HBM's bandwidth increase comes from stacking multiple DRAM chips in a single module, providing numerous memory channels for parallel access. This advantage presents new opportunities for enhancing the performance of graph processing accelerators. However, there are many challenges in efficiently utilizing the high bandwidth to handle irregular memory accesses during graph processing.

Specially, each edge in the Compress Sparse Row (CSR) format occupies only a narrow bit width (e.g., 32-bit) to record the destination ID. This bit width is inconsistent with HBM's 256-bit prefetch length for each memory transaction. HBM's adoption of a 32-bit AXI access mode results in extremely low bandwidth utilization. Additionally, cross-channel access by the HBM subsystem on FPGAs introduces the problem of internal contention, further increasing the delay and degrading the bandwidth utilization. Hence, it is imperative to implement efficient and economical routing among multiple HBM memory channels. This requires utilizing the finite logical resources available on FPGAs while maximizing memory bandwidth utilization.

In this subsection, we present ScalaBFS2[17], which is designed for BFS, the basis of the graph algorithm. ScalaBFS2 focuses on achieving the highest performance of BFS on a single HBM-enhanced FPGA chip. As shown in Fig.4, ScalaBFS2 contains several PEs that can communicate with each other through elaborate crossbars.

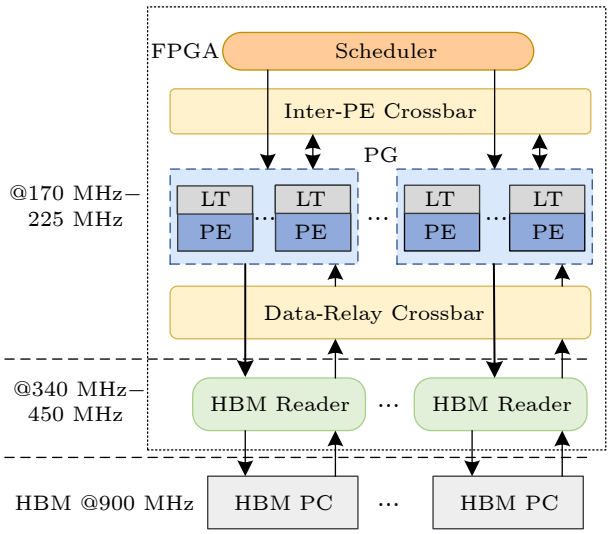


Fig.4. Overall architecture of ScalaBFS2. PG: processing group; LT: local store.

To improve the efficiency of HBM memory access, ScalaBFS2 is committed to maximizing bandwidth utilization by engaging as many HBM PCs as possible. By designing an independent HBM reader responsible for reading data from the corresponding HBM PC, ScalaBFS2 separates memory access circuits from the PEs that execute the BFS algorithm. This strategy addresses the mismatch between the long bit-width of the HBM PCs and the narrow representation of graph data while maintaining a lean design for resource conservation. Specifically, the HBM reader operates within the frequency range of [340 MHz, 450 MHz], twice the frequency of the PEs. Moreover, it accesses data with the same bit-width as the HBM prefetch length (i.e., 256-bit), enabling high-speed access to HBM. Upon receiving responses from the HBM PCs, the HBM reader extracts valid data using a filter and passes it to the subsequent process (i.e., crossbar) round-robin, ensuring a well-balanced load for subsequent processing.

Furthermore, ScalaBFS2 proposes new crossbars for data transfer to save resources. Assuming N PEs are constructed in ScalaBFS2, a full crossbar will consume N^2 FIFOs, resulting in serious resource consumption. ScalaBFS2 factorizes N and constructs an equivalent multi-layer crossbar that performs data

transfers in a pipelined manner. The number of FIFOs utilized internally in the multi-level crossbar is significantly reduced compared with the full crossbar. This approach significantly reduces resource consumption while maintaining the performance of the BFS algorithm.

We perform an evaluation of ScalaBFS2 on the XCU280 chip. The experiment evidences that ScalaBFS2 can utilize all 32 PCs and build up to 128 PEs on the XCU280. ScalaBFS2[17] achieves a peak performance of 56.92 GTEPS, which is 2.52x–4.40x speedup over the latest graph processor (i.e., ReGraph[32]) built on the same device. ScalaBFS2 has 1.34x–2.40x speedup over Gunrock running on the A100.

2.3 ScalaGraph: Scaled to Thousands Cores

HBM provides high memory bandwidth while existing graph accelerators fail to utilize its potential fully. Specifically, a graph accelerator running at 250 MHz needs at least 1 024 ((1 024 GB/s)/(0.25 GHz ×4 B)) PEs for edge computation to exhaust HBM bandwidth, assuming that an edge is 4 bytes in size. However, existing work employs a centralized PE interconnection[33], which cannot scale to more than 256 PEs from our experimental results.

The gap between large-scale PEs and HBM results in untapped hardware potential. To address the new bottleneck caused by HBM, it is critical to prioritize the scalability of memory hierarchy over the performance efficiency of the individual PE. In other words, graph processing accelerators should achieve a better balance between the hardware overhead of interconnect architectures and the communication efficiency among PEs.

To achieve the above goal, we propose ScalaGraph[13], a graph processing accelerator based on a distributed on-chip memory hierarchy. As shown in Fig.5, ScalaGraph consists of multiple tiles that are interconnected through an on-chip network (NoC). Tiles contain three key modules: Prefetcher, Dispatcher, and Processor. Unlike the centralized on-chip memory hierarchy, each HBM block in ScalaGraph is connected to one PE. A PE communicates with another through NoC, which improves the scalability of the accelerator by avoiding large-scale communica-

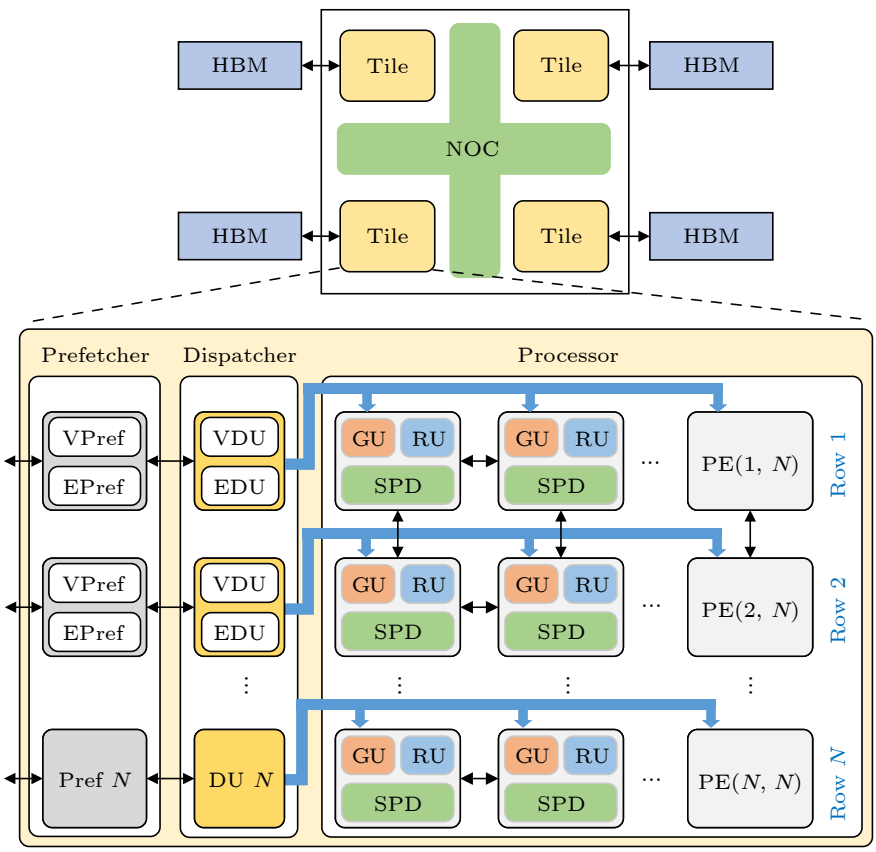


Fig.5. ScalaGraph architecture[13]. Pref: prefetcher, VPref: vertex prefetcher, EPref: edge prefetcher, EDU: edge dispathing unit, DU: dispathing unit, SPD: scratchpad, RU: routing unit, GU: graph unit, and VDU: vertex dispatching unit.

tion among PEs.

While ScalaGraph improves scalability, it brings two challenges. First, the distributed on-chip memory hierarchy significantly increases communication latency and volume. Specifically, communication among PEs may involve multiple route transmissions, which can increase the delay of PE communication. Second, the power-law degree distribution can lead to severe load imbalance in a distributed environment. A few vertices are connected to most edges, causing some PEs to be busier than others.

For the first challenge, ScalaGraph proposes a software-hardware co-design approach to address it. On the software side, ScalaGraph proposes an architecture-aware data mapping mechanism called Row-Oriented Mapping. This mechanism maps edge workloads to different PEs based on the source vertex's row ID and the target vertex's column ID. By doing so, it effectively eliminates inter-column communication among PEs and minimizes the communication latency in each PE row. In hardware, ScalaGraph designs a routing unit. Specifically, ScalaGraph adopts the idea of parallel accumulation[9] and provides four stages to minimize the communication latency in each PE column.

For the second challenge, ScalaGraph implements a degree-aware scheduling mechanism, which leverages the observation that memory addresses of active vertices in graph processing tend to be contiguous. This scheduling mechanism ensures a balanced distribution of workloads during the scatter phase. To enhance load balancing during the apply phase, ScalaGraph adopts a novel pipelining architecture, inter-phase pipelining, to reduce idle PEs. In this pipeline model, the updated results in the apply phase are immediately sent to the dispatcher module, allowing the scatter phase to process without waiting for the entire active vertex list to complete. This avoids load imbalance caused by synchronization in the apply phase and enhances processing efficiency.

We implement ScalaGraph and perform evaluations and experiments on a Xilinx Alveo U280 accelerator card. As shown in Fig.6, our experiments on classical datasets show that ScalaGraph supports scaling beyond 1 024 PEs. Furthermore, ScalaGraph outperforms state-of-the-art accelerators[33] by 2.2x.

3 Graph Software System

To fully explore the potential of large-scale graph processing, we develop graph processing systems that exploit the characteristics of different hardware architectures. Specifically, we introduce HotGraph, GraphFly, and CGraph for CPU, which focus on accelerating static, dynamic, and concurrent graph processing, respectively. Additionally, we present DiGraph, an iterative directed graph system designed for GPU architectures. This system utilizes vertex dependencies to minimize the number of graph iterations required. Furthermore, we propose GraphSu for FPGA, which is a library for dynamic graph processing. Lastly, we propose FBSGraph for distributed architectures, which enables fast convergence of asynchronous graph processing.

3.1 Graph Processing on CPU

In this subsection, a CPU-based static, a CPU-based dynamic, and a CPU-based concurrent graph processing system are presented respectively.

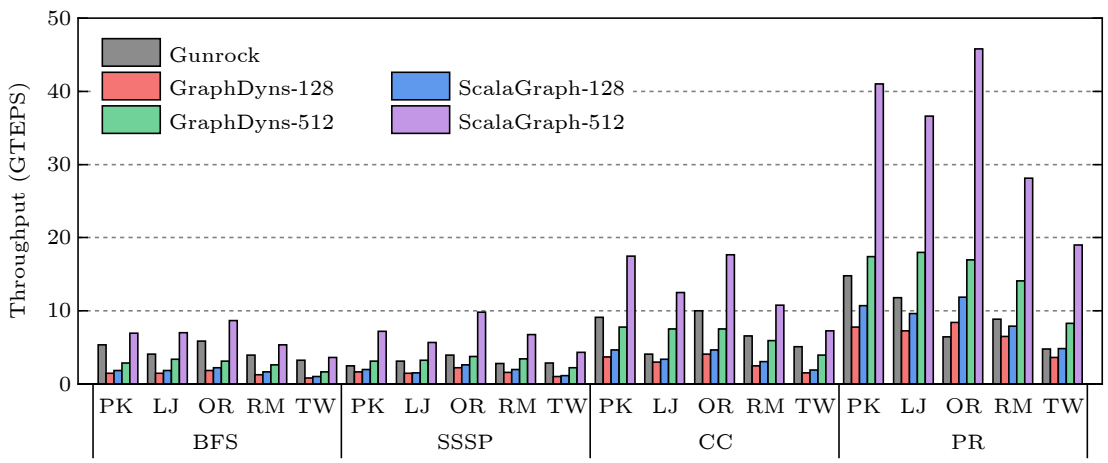


Fig.6. Throughput[13] of ScalaGraph vs GraphDyns and Gunrock. CC: connected components; PK: Pokec; OR: Orkut; RM: RMAT[24]; TW: Twitter.

3.1.1 Static Graph Processing on CPU

Asynchronous graph processing is more efficient than synchronous graph processing. Therefore, many asynchronous processing systems[34] have emerged. However, existing asynchronous graph processing systems ignore random state propagation between separating partitions. This omission leads to expensive overhead and seriously decreases the convergence speed of graph algorithms.

We propose a graph processing system called HotGraph[18], inspired by the cascade effect to tackle the bottleneck above. This graph processing system builds a backbone structure called Hotgraph, which comprises hot vertices and the paths between vertices. By leveraging this backbone, instead of waiting for all vertices in the local partition to complete their state updates, vertex states can be quickly propagated to neighboring graph partitions, thereby mitigating the cross-graph partitioning overhead.

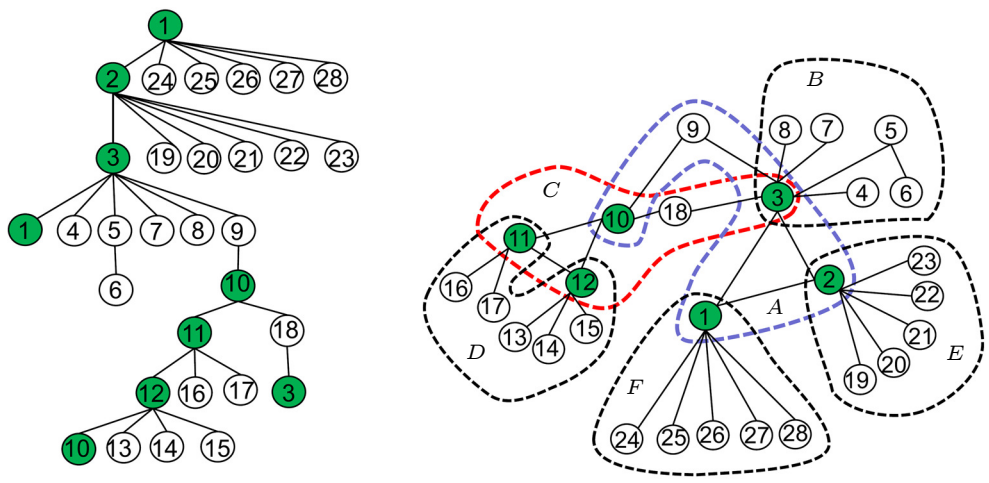
In particular, we present a novel algorithm for constructing HotGraphs in large graphs. The algorithm starts by partitioning the graphs into subgraphs using the vertex-cut graph partitioning algorithm[35] and parallelly extracting a set of hot vertices within the subgraphs. Then, we select a hot vertex as the root and traverse all edges in the subgraphs using a depth-first order to establish paths between the hot vertices. Next, we build connections between subgraphs using hot vertices and associated paths. These hot vertices and valid paths between them are classified as HotGraph, while the remaining vertices and edges are classified as cold partitions. Fig.7 shows the example of HotGraph extraction. Finally, to ensure swift transmission of vertex state information among subgraphs, we prioritize HotGraph with high importance.

Our experiments investigate the performance of HotGraph compared with Maiter[34]. It shows that HotGraph achieves an 80.8% reduction in execution time, and the execution time decreases as the available memory size increases. These prove that HotGraph accelerates vertex state transmission for asynchronous graph processing in shared memory systems.

3.1.2 Dynamic Graph Processing on CPU

Dynamic graph processing is a crucial component in analyzing dynamically changing graph data in real-time. However, real-time updates to graphs present difficulties for current graph processing systems. Various systems for dynamic graph processing have been introduced to tackle the issues caused by the swift updates in graph data. Examples include KickStarter[36] and GraphBolt[37]. These systems utilize incremental computing techniques to perform real-time data analysis by leveraging the results of previous computations.

Incremental computation reduces response time, but reusing previous results may lead to inaccuracies. KickStarter[36] and GraphBolt[37] use a refinement-computation model to ensure correctness, but the model introduces redundancies in memory accesses. Redundancies arise from two sources: 1) refined vertex values are written to memory and then retrieved for recomputation on affected vertices, and 2) edges of these affected vertices are initially traversed to identify refinement effects and then reaccessed during re-



(a) (b)

Fig.7. Example illustrating how to extract a hot graph[18]. (a) Depth-first search. (b) Partitions generated.

computation to propagate computed values. In our profiling analysis of GraphBolt, as depicted in Fig.8, we observe that redundant memory accesses consume an average of 68% of the runtime in both phases.

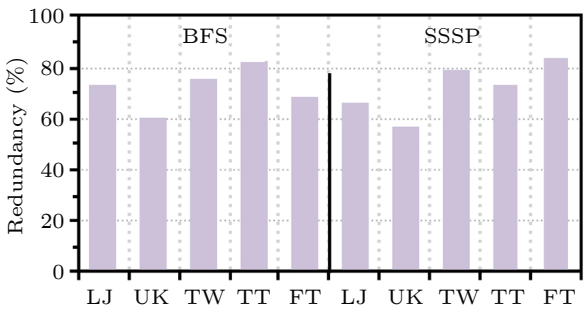


Fig.8. Redundant memory accesses ratio[19]. UK: UKDomin; TT: Twitter MPI; FT: Friendster.

In this work, we propose GraphFly[19], a solution to address the redundant memory access by bridging the gap between refinement and recomputation during real-time edge updates. GraphFly introduces a dependency flow based approach to graph processing. Using the dependencies among dynamic updates, GraphFly performs refinement and recomputation asynchronously, thus significantly reducing redundant memory access. To accomplish these goals, GraphFly includes two essential modules.

Dependency Management. This module initially creates dependency flows using D-trees and effectively organizes them in a spatio-temporal scheduling sequence. D-trees are based on the elimination tree theory and depict the dependency connections among vertices in a lower triangle matrix as a forest. By dividing these D-trees at their root vertices, we extract dependency flows. These flows derived from the lower triangle matrix capture the vertex dependencies in the structural space. To ensure correctness, these flows must be executed in a specific order at runtime, constrained by the upper triangle matrix. Subsequently, these commands for coordinated spatio-temporal scheduling are provided to the processing engine for execution.

Processing Engine. GraphFly achieves its efficiency by combining expert storage management strategies that focus on data locality, along with a parallel asynchronous processing model that relies on dependency flows. In cases where vertex data needs to be accessed within these dependency flows, the data might be distributed across various storage regions. To enhance memory access efficiency, GraphFly introduces a storage format specifically tailored for dependency flows. This format allows for the efficient and compact storage of vertex data in memory, which in turn reduces memory access overhead and improves overall efficiency. Furthermore, dynamic updates can be distributed across different dependency flows. GraphFly loads and executes dependency flows asynchronously, adhering to the predefined scheduling order.

Our experiments[19] show that GraphFly outperforms state-of-the-art systems KickStarter and GraphBolt by 5.81x and 1.78x on average, respectively.

3.1.3 *Concurrent Graph Processing on CPU*

In recent years, the application of Concurrent Iterative Graph Processing (CGP) jobs in social network analysis, bioinformatics, and recommendation systems has experienced significant growth. However, deploying multiple CGP jobs on the same graph using existing systems presents several challenges. One of the critical issues is the repetitive loading of graphs into the cache by different jobs at different times, leading to expensive data access and low throughput. Furthermore, many CGP jobs exhibit strong temporal and spatial correlations, resulting in substantial redundant data access.

Numerous researchers are striving to optimize data access and computational scheduling to improve the throughput of CGP jobs. Certain methods have been employed, such as exploiting high sequential memory bandwidth, data locality, and efficient redundant data processing. However, these efforts tend to focus on specific aspects and may struggle to provide comprehensive solutions for the numerous challenges.

We introduce CGraph[20], a system designed to improve the throughput of CGP jobs through a data-centric Load-Triggered Push (LTP) model. The LTP model effectively separates the graph structure data from the corresponding vertex state. It streams the shared graph structure partitions in the cache and triggers parallel processing of relevant jobs. Then, vertex states are pushed for efficient convergence. This method reduces data access costs by processing multiple jobs in a common order through amortizing access to shared subgraphs. Furthermore, the ability to utilize the shared graph structure data for multiple jobs leads to a reduction in both cache usage and memory consumption.

The LTP model follows these steps. Firstly, the CGP jobs load the shared graph structure partition in

a specific order. Secondly, the relevant jobs are triggered for each loaded partition, allowing them to perform concurrent graph processing operations. Thirdly, a state update is executed to update the vertex states associated with that partition after each job completes its graph processing operations. Once all the relevant jobs finish processing a partition, the updated state of the partition is pushed to ensure state synchronization across vertices in different partitions. Lastly, once all active partitions have been processed within the current iteration, each job proceeds to initiate a new iteration. Different CGP jobs may be at varying iterations of their graph processing. By utilizing state pushing, jobs can maintain consistency across various iterations.

In order to optimize throughput, we introduce a scheduling algorithm that leverages the core-subgraph. This algorithm strategically determines the loading order of partitions, effectively enhancing overall performance. Initially, we identify a core subgraph within the graph, comprising core vertices and the path edges between vertices. Next, we uniformly divide the graph based on the identified core subgraph. Specifically, we group the edges of the core subgraph into several partitions of equal size. The remaining edges are equally allocated to other partitions. By frequently loading and processing the core vertices, we minimize the cost of loading early-converged vertices within the same partition. This approach effectively reduces the required bandwidth and optimizes cache space consumption.

In contrast to other solutions, the CGraph approach provides a notable 2.31 times increase in throughput for CGP jobs. This improved performance is mainly due to the lower average data access cost.

3.2 Graph Processing on GPU

Previous research has extensively studied iterative directed graphs due to their broad applications in real-world scenarios[38, 39]. However, significant challenges still exist in processing iterative directed graphs on GPUs. Particularly, the vertices within each directed path are concurrently processed by multiple GPU threads and update their states in each iteration based on the previous states of their forward neighbors. If the dependencies of vertex updates are unknown, the new states of active vertices must be propagated to subsequent vertices over multiple iterations. This problem affects the convergence speed and incurs a higher cost regarding vertex state loading.

Fortunately, through numerous experiments, we obtain two fundamental observations that reduce the number of redundant vertex updates in the iterative-directed graph algorithm. Firstly, when vertices are processed asynchronously and sequentially along the directed path in a round, the new states of the vertices can be utilized to process other vertices within the same round. Secondly, a significant number of directed edges do not form loops, thereby enabling further reduction of redundant processing.

Based on these observations, we propose an efficient GPU-based iterative-directed graph processing system called DiGraph[21]. DiGraph introduces three innovative methods to leverage vertex dependencies (in Fig.9). First, DiGraph decomposes a directed graph into disjoint directed paths, treating them as fundamental units for parallel processing. This approach facilitates the propagation of vertex states along the directed paths, leading to improved convergence speed and better data utilization. Second, DiGraph assigns these paths to the GPUs for parallel processing based on the graph's topology. The system reduces the overhead associated with reprocessing by processing numerous paths only once and converging them in topological order. Third, DiGraph introduces a path scheduling strategy on streaming multiprocessors to exploit the vertex dependencies between different paths fully.

However, the implementation of DiGraph faces

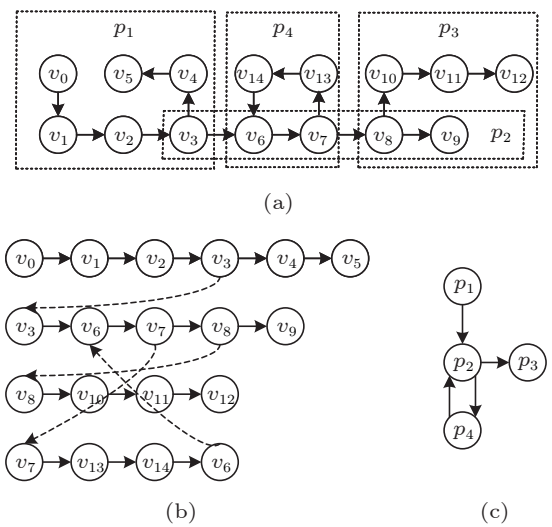


Fig.9. Example of how to partition a directed graph into directed paths and obtain a dependency graph, where p_i denotes the i-th path and v_i denotes the i-th vertex[21]. (a) Example of a directed graph divided into four paths. (b) Diagram of dependency relationships between paths. (c) Path dependency graph.

two challenges. The first challenge is the skewness in the lengths of the generated paths. Since GPU threads within a warp execute instructions in lock-step, imbalanced loading can occur if the lengths of the paths vary significantly. To address this challenge, ensuring that each GPU thread processes an approximately equal number of edges is essential. Therefore, if one GPU thread is assigned a long path for processing, other GPU threads are assigned multiple shorter paths to maintain load balance. The second challenge is the presence of dependencies between the directed paths, which causes some paths to be reprocessed. To tackle this problem, Digraph constructs a directed acyclic graph (DAG) for the dependency graph of the paths. According to DAG's topological order, it allows the asynchronous dispatch of the paths to GPUs with low reprocessing costs. In detail, it tries to asynchronously dispatch the paths to GPUs for parallel processing layer by layer.

Our experiments demonstrate that DiGraph significantly improves performance over Gunrock[38] and Groute[39]. Specifically, DiGraph achieves performance gains ranging from 2.25 times to 7.39 times over Gunrock and from 1.59 times to 3.54 times over Groute. Furthermore, when the number of GPUs is increased from 1 to 4, DiGraph shows a significant 62.9% reduction in graph processing time. This surpasses Gunrock's 46.3% reduction and Groute's 56.5% reduction, demonstrating DiGraph's superior scalability for larger graph processing tasks.

3.3 Graph Processing Based on FPGA

Graph processing is a powerful technique for analyzing relationships in various domains. Hardware acceleration has significantly enhanced graph processing performance in the past decade. FPGA is a promising candidate for graph processing among the different hardware platforms due to its parallelism, power consumption, and flexibility advantages. Therefore, substantial work[9] has been proposed on FPGA to improve graph processing performance.

Most graph accelerators based on FPGA are limited to static graphs, which do not change over time. However, real-world graphs are often dynamic, meaning they can change over time. Processing dynamic graphs requires two fundamental processes: graph computation and graph update. In previous work, graph computation has been investigated well, while the graph update has many problems to be addressed. In actuality, graph update is as essential as graph computation.

To bridge the gap between the two processes, we design a graph update library[22], which could be readily incorporated into accelerators for the static graph to process the dynamic graph. There are other challenges in achieving high-efficiency graph updates. Due to abundant real-world graph updates, PEs may cause off-chip communication overheads for retrieving off-chip edge data. We observe that the actual dynamic graph has spatial similarity. Thus, we apply differential data management to transform most random off-chip edge access into on-chip access.

While differential data management fits well with spatial similarity, there are at least two challenges. First, we need to precisely determine the value of each vertex to assign its related edges to certain memory devices. However, it is challenging to measure the vertex value precisely as it keeps changing. Second, applying the differential memory, on-chip and off-chip memories both have duplicates with edge data. It is challenging to ensure data location efficiently and precisely.

To achieve these goals, we present GraSU[22], an FPGA-based graph update library. GraSU is effective in utilizing spatial similarity. The architecture of GraSU is shown in Fig.10. It comprises five parts: dynamic graph storage, incremental value measurer, edge updates dispatcher, edge updates handling logic, and value-aware memory manager. We offer interfaces of GraSU for programmers so that GraSU can be incorporated into static graph accelerators without modifying the graph algorithm code.

For the first challenge, GraSU proposes an equation to measure the vertex value. The equation dynamically quantifies vertex values. We overlap the value measurement and graph computation to hide the measurement overhead. For the second challenge, GraSU implements value-aware memory access and a high-value data identification mechanism. We choose UltraRAM to store data with high value for its coarse-grained feature. We implement a bitmap-based approach for balancing memory access performance and space consumption well.

We implement GraSU in Verilog and integrate it into AccuGraph with only 11 lines of code modification. We perform evaluations on a Xilinx Alveo U250 card[22]. Compared with two state-of-the-art CPU-based dynamic graph systems, Stinger and Aspen, the update throughput of GraSU is 34.24x and 4.42x higher on average. GraSU achieves an average increase of 9.80x and 3.07x in overall latency.

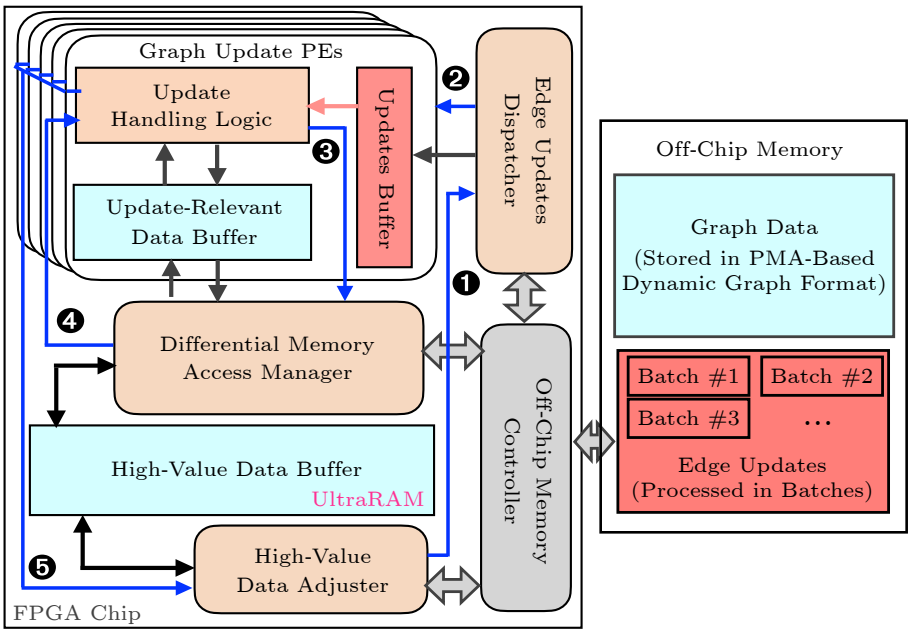


Fig.10. GraSU architecture[22].

3.4 Distributed Graph Processing

Distributed graph processing with the asynchronous (AGP) paradigm faces challenges in efficiently propagating vertex states between vertices for two reasons. First, current graph partitioning methods separate vertices along a path into different partitions for load balancing. Consequently, vertices initially connected on the same path cannot complete their processing within a single iteration. In other words, multiple iterations are required to complete the information propagation of all vertices on a path. Second, the current execution modes of AGP, such as round-robin and prioritized modes, have limitations in efficient vertex state propagation. In the round-robin mode, vertex states cannot propagate along the path. On the other hand, the prioritized mode incurs a significant amount of additional overhead for propagation.

After careful analysis, we obtain the following two observations. First, the culprit of these problems is that the path information of the graph is not considered when determining the order of state propagation. Second, leveraging the cascading effect is the key to resolving this problem. When processing the graph, if vertices are sequentially processed along their paths, any changes in their state will promptly impact other vertices on the same path. This means that by updating vertices along the path, the new state of a vertice can be rapidly propagated to subsequent vertices.

Based on these observations, we propose the Forward and Backward Sweeping (FBS) execution paradigm[23] shown in Fig.11. Further, we develop an asynchronous graph processing system named FBSGraph[23], specifically designed for distributed platforms. In FBSGraph, we divide the graph into disjoint paths. Vertices along each path are processed sequentially in the order defined by the path, with alternating forward and backward directions. This paradigm guarantees that vertex information is propagated to both forward and backward neighbors.

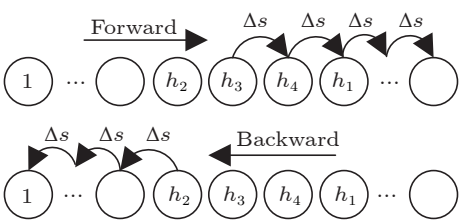


Fig.11. Forward and backward sweeping mode. Forward and backward propagation follows the order of vertice h_i in the partition[23].

In distributed architectures, each node processes multiple paths in the graph. However, it causes significant communication overhead due to the dependencies between these paths. To minimize this overhead, FBSGraph incorporates a local buffering mechanism for vertex state changes targeting the same remote node. The alterations in states are subsequently sent to the remote node at set time intervals. When these intervals are too brief, it results in frequent transfers of state values, thereby increasing communication overhead. Conversely, when the intervals are exces-

sively long, the changes in the state of the current vertex are stored for an extended period, leading to a slower convergence. In order to strike a balance between the communication cost and convergence speed, we have introduced the concepts of automatic time intervals and FBS-Round time intervals, which assist in determining the most suitable timing intervals. We also offer a number of optimization schemes, as detailed in FBSGraph[23].

When analyzing the total execution times of FBSGraph-SPO-1, Maiter-Pri, and Maiter-RR on a classical dataset, it is evident that FBSGraph-SPO-1 significantly outperforms Maiter-RR by reducing the execution time by 39.7%. Additionally, experiments carried out on a 1 024-core cluster demonstrate the excellent scalability of our approach.

4 Emerging Hardware/Software Co-Designs and Beyond

The rapid advancements in Big Data and Artificial Intelligence have introduced new challenges for traditional graph processing. First, the relationships between data have become more complex, resulting in valuable information embedded in these associations. To extract meaningful information from complex graph data, novel graph processing workloads have emerged, including graph construction, hypergraph processing, and heterogeneous graph processing. Consequently, the requirements for graph processing have become increasingly diverse and complex.

Furthermore, emerging hardware technologies, including Processing In/Near Memory Architecture (PIM/PNM), offer significant computing and memory resources that can alleviate the bottlenecks in graph processing. PIM/PNM architectures can reduce data loading latency, making them well-suited to address the challenges of graph processing. These hardware advancements provide new avenues for improving the efficiency and performance of graph processing workloads. However, effectively utilizing these hardware technologies can be a non-trivial task.

In this section, we present our contributions to processing complex graph workloads and efficiently utilizing emerging hardware architectures.

4.1 Emerging Graph Processing

This subsection introduces our work on emerging graph processing in the order of graph construction, hypergraphs, and heterogeneous graphs.

4.1.1 Graph Construction

The k-nearest neighbor graph (kNNG) is widely used in databases, large language models, and other domains. However, constructing an exact kNNG is time-consuming, with a time complexity of $O(dn^2)$, where d represents the dimensionality of the feature vectors and n denotes the total number of vertices. When constructing large-scale kNNG, the cost becomes unacceptable. Fortunately, many applications can tolerate some loss of accuracy, allowing a significant reduction in construction cost by adopting approximate graph structures. As a result, recent research has focused on efficient methods for constructing approximate kNNGs. Among these methods, NN-descent[40] has emerged as the most popular approach.

The NN-descent algorithm optimizes the construction method based on the idea that "My neighbor's neighbor may also be my neighbor." It starts by randomly initializing neighbors for each vertex and then iteratively refines the graph structure until convergence. The iteration consists of three stages: sampling, computation, and update. During the sampling stage, a subset of neighbors is sampled for each vertex, effectively reducing redundant computations. The computation stage begins once the sampling stage is completed for all vertices. In this phase, the algorithm computes the similarity between the sampled neighbors of each vertex. Finally, in the update phase, the algorithm updates the neighbor lists of the corresponding vertices based on the computed similarity values.

We analyze the time distribution of the three stages in NN-descent and find that the computation stage is the primary bottleneck in the algorithm. This stage involves a significant amount of high-dimensional vector computation. In particular, the computation stage is not efficient enough in terms of computation and memory access. First, the phase sequentially processes the source vertices and computes the similarity between the target neighbors in each source vertex. Unfortunately, this sequential vertex scheduling scheme leads to many random memory accesses due to the low overlap between sequential vertex neighborhoods. Moreover, many vector computations become useless, resulting in a severe waste of computational resources.

To address the above challenges, we design a hardware accelerator called FNNG[24], as described in Fig.12. FNNG comprises three modules: the sampling, computation, and update modules, which correspond

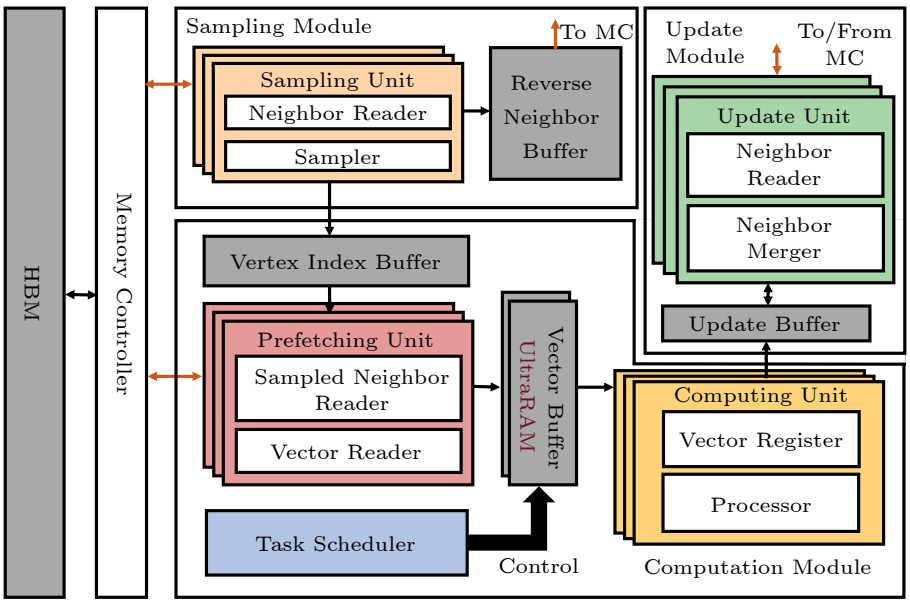


Fig.12. Architecture of FNNG[24].

to NN-descent's three stages. Our accelerator design is based on two observations. First, vertices in close spatial proximity often share parts of neighborhoods. Therefore, we introduce a block-based scheduling method that prioritizes the processing of vertices within a specific spatial region. This approach enhances data reuse and reduces off-chip memory accesses. In addition, a lot of vector computations do not require precise results. Therefore, we propose the Useless Computation Aborting method to terminate unnecessary computations for certain dimensions of feature vectors, resulting in significant time savings.

We evaluate FNNG's performance on five classic datasets and validate it on the Xilinx Alveo U280 accelerator card[25]. The experiments show that our solution achieves approximately 190x and 2.1x performance improvements over the latest CPU and GPU solutions, respectively, while maintaining similar graph accuracy.

4.1.2 *Hypergraph Processing*

A hypergraph is a complex graph model that allows edges to connect multiple vertices, making it capable of representing complex multilateral relationships among numerous entities. As it is powerful to capture intricate relationships, various fields widely adopt hypergraphs for data analysis, which urges efforts to enhance the performance of hypergraph processing techniques.

The early research used index-ordered scheduling to execute bipartite edge tasks. Index-ordered scheduling processes the active hyperedges in the order of indices. While the scheduling is easy to understand, it typically results in a high cache miss rate. ChGraph[41], one of the best hypergraph systems, exploits the vertex locality between two hyperedges and takes chains to schedule the order of hyperedges. However, ChGraph cannot exploit more complex inter-chain locality. Thus, the scheduling will lead to redundant access and bandwidth underutilization. Those two issues cause memory subsystem inefficiency.

To take advantage of data locality, we present a data-centric Load-Trigger-Reduce (LTR) execution model, which is difficult in task-centric hyperedge processing systems. We decompose the computation phases into three processes: hypergraph loading, task execution triggering, and temporary value reduction. In a data-centric model, once vertex data is loaded, corresponding tasks are invoked.

While LTR brings benefits in data locality, its implementation still has several challenges. In the load step, due to recurrent intersection, operations will cause much inefficient memory access. Worse, the static vertex data partition strategy still leads to excessive data loading. In the reduce step, heavy atomic protection overhead is introduced to resolve data conflicts due to multiple simultaneous updates to the same data. Although the CPU can benefit from our data-centric model, its performance is constrained by the architecture.

The above discussions prompt us to design the hardware accelerator, XuLin[25]. As shown in Fig.13, XuLin consists of five parts: Loader, Translator, Trig-

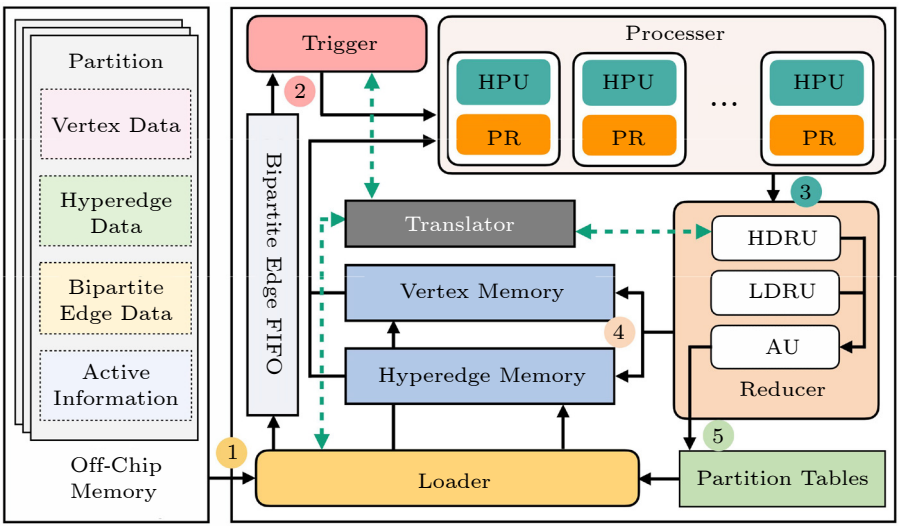


Fig.13. XuLin architecture[25]. HPU: hypergraph pocessing unit, PR: private register, AU: activation unit, HDRU: high-priority data reducing unit, and LDRU: low-priority data reducing unit.

ger, Processor, and Reducer. XuLin uses a key-value table to eliminate intersection operations. XuLin also takes adaptive data loading and chunk merging mechanisms to reduce data transfer. We advocate priority-based differential data reduction hardware, which can minimize the cost of resolving data conflicts.

We perform our evaluations and experiments on a Xilinx Alveo U250 card and a simulator. As shown in Fig.14, compared with Hygra and ChGraph on five graphs with four graph algorithms, the running time of XuLin is 20.47x and 8.77x shorter on average, respectively.

4.1.3 *Heterogeneous Graph Processing*

Existing computer architecture research for graph neural networks has mainly focused on homogeneous graphs containing only one vertex and edge type. However, in real life, heterogeneous graphs containing multiple vertex and edge types are much more common and can better represent information than homogeneous graphs. The field of graph neural networks focusing on heterogeneous graphs has many problems that have yet to receive much attention. This subsection presents our efforts in accelerating metapath-based heterogeneous graph neural networks (HGNNs).

Metapath is a critical concept in heterogeneous graph neural networks to represent an ordered sequence of different types of vertices. Fig.15 depicts an example of an academic heterogeneous graph, where Author-Paper-Conference-Paper-Author (APCPA) and Author-Paper-Author (APA) are the defined metapaths expressing different semantic information, APA expresses the two co-authors of a paper, and APCPA indicates both authors have presented papers at the same conference. Compared with traditional graph neural networks that aggregate information directly based on neighboring vertices, HGNNs are more complex to aggregate information based on metapaths.

HGNNs are effective for representing information, but they face several performance bottlenecks. First, the preprocessing phase of HGNNs involves metap-

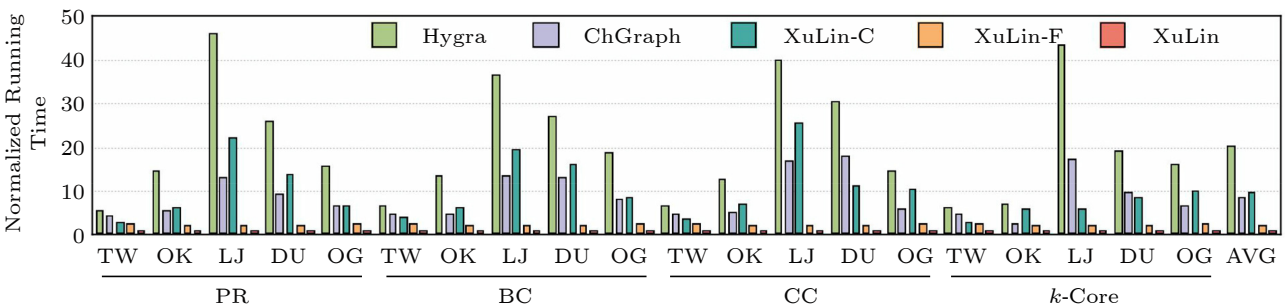


Fig.14. Running time of XuLin[25]. TW: trec-wt; OK: com-Orkut; DU: delicious-ut; OG: Orkut-group.

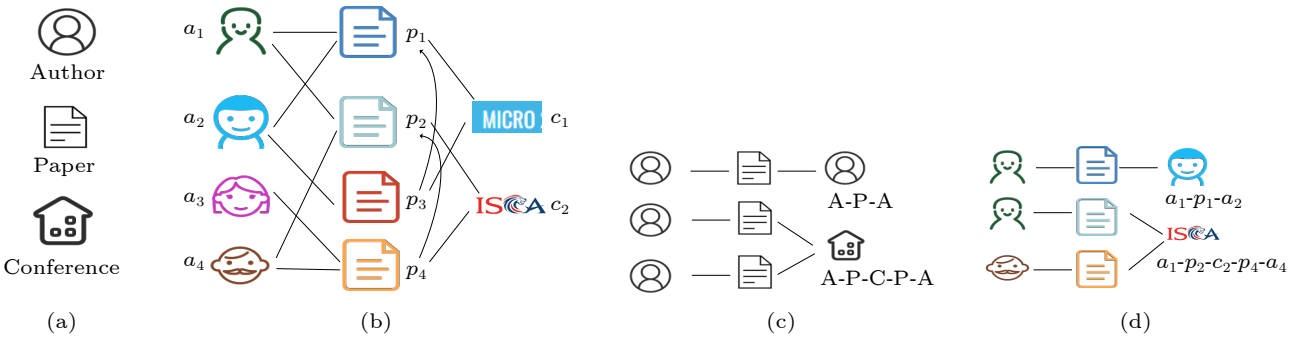


Fig.15. Illustrative example of a heterogeneous graph[26]. (a) Vertices' three types. (b) Academic heterogeneous graph containing three types of vertices and three types of connections, where a_i means the i-th author, p_i means the i-th paper, and c_i means the i-th conference. (c) Two defined metapaths (i.e., APA and APCPA). (d) Two metapath instances.

ath instance matching, which generates multiple instances starting from a single vertex. This phase can be time-consuming and requires significant storage overhead. Second, vertex feature aggregation along metapath instances involves redundant computations due to multiple identical vertices among instances. This results in repeated aggregations. Finally, HGNNs encounter memory bottlenecks due to irregular memory accesses. This motivates us to leverage near-memory processing to overcome the memory bottleneck of HGNNs, which offers high memory bandwidth, low data access latency, and lower energy consumption[8, 10].

We propose MetaNMP[26] for HGNNs to address the above bottlenecks. MetaNMP introduces a novel computational paradigm called "cartesian product-like" to complete metapath instance matching efficiently at runtime. It eliminates the need for a preprocessing phase and avoids the overhead of storing metapath instances. Moreover, most of the redundant computations among meta-path instances are derived from that these instances come from the same vertex. Instead of independently aggregating each metapath instance, MetaNMP aggregates vertex features along the direction of metapath instance derivation, minimizing redundant computations. Finally, MetaNMP incorporates two customized modules as near-memory computing units to accelerate HGNNs. The metapath instance management module, integrated at the DIMM-level, generates metapath instances using the cartesian product and monitors redundant computations. The feature aggregation module, integrated at the rank level, handles feature aggregation based on metapath instances.

Fig.16 displays MetaNMP's running time. Compared with CPU, GPU, HyGCN, AWB-GCN, and RecNMP, MetaNMP achieves 4 225.51x, 415.18x, 48.96x, 78.34x and 17.23x improvements, respectively.

4.2 Processing In/Near Memory

This subsection describes the design of graph analysis and graph learning on ReRAM and then describes graph processing's design on CMOS.

4.2.1 *ReRAM-Based Graph Processing*

Resistive Random Access Memory (ReRAM) is an exceptionally promising non-volatile memory that achieves data storage through resistance alteration[42]. It consists of individual cells with a metal-insulator-metal structure and an oxide layer between electrodes[43]. These cells are arranged in a crossbar configuration to optimize spatial utilization, allowing for in-situ Matrix-Vector Multiplication (MVM) oper-

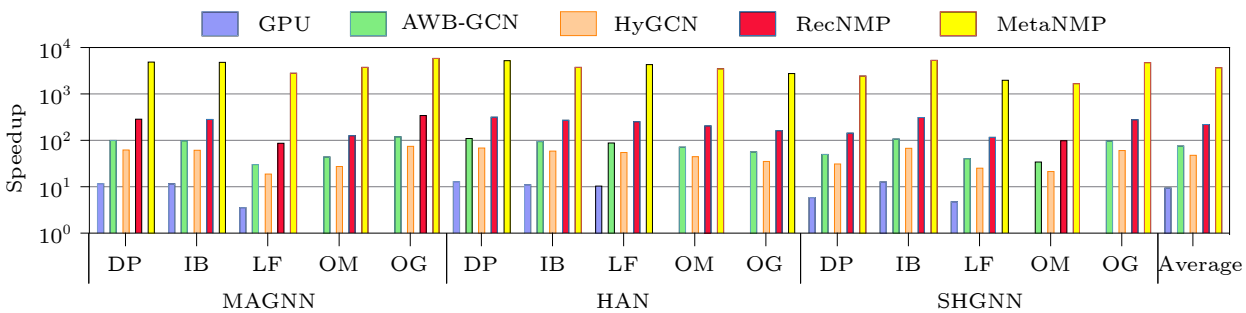


Fig.16. MetaNMP's running time[26]. DP: DBLP; IB: IMDB; LF: LastFM; OM: OGB-MAG; OG: OAG.

ations[43]. Specifically, once the input data vector is loaded onto the crossbar's wordlines, the corresponding output results can be computed quickly within a single cycle using the bitlines[42].

Graph analysis applications can be viewed as MVM computations[44], offering the opportunity for accelerated processing using ReRAM. In this work, Spara[27] introduces a novel graph analysis accelerator that harnesses the advantages of ReRAM and effectively utilizes parallelism among memory banks.

To exploit inter-bank parallelism, Spara employs the Random-Wordline-Consecutive-Bitline (RWCB) vertex mapping scheme. This approach divides edges into separate groups and assigns each group to a specific memory bank, preventing target vertices from overlapping within a bank. Spara introduces two methods to support this vertex mapping: the Crossbar-Bounded Graph Reordering method, which optimizes crossbar population, and the Wordline-Cut Reorganization method, which handles sparsity issues caused by vertex activation.

Within each memory bank, Spara adopts a hybrid vertex mapping strategy that incorporates both RWCB and random-wordline-random-bitline (RWRB) mappings, thereby harnessing the advantages offered by both mapping schemes. To accomplish this, Spara designs a wordline dispatcher that dynamically detects the density of each wordline. Wordlines with low density are temporarily stored in a function array during runtime and processed based on the RWRB vertex mapping scheme. On the other hand, dense wordlines are directly loaded into the crossbar and aggregated using the RWCB vertex mapping scheme, thereby striking an optimal balance between cost and efficiency.

4.2.2 *ReRAM-Based Graph Learning*

Graph Convolutional Network (GCN) has recently gained considerable attention among researchers[45–47]. Like graph analysis, GCN's combination and aggregation stages can be effectively regarded as MVM computations[48], rendering it a suitable candidate for acceleration using ReRAM. An intuitive strategy entails leveraging existing ReRAM-based neural network or graph analysis techniques to accelerate GCN's combination and aggregation stages.

However, applying these techniques still presents challenges. First, GCN requires simultaneous consideration of dense weight matrices and sparse graph data during processing. However, directly mapping sparse graph data onto the crossbar results in a significant number of idle cells, thereby hindering the achievement of optimal hardware utilization. Second, compared with traditional graph data, GCN exhibits higher-dimensional vertex features, typically ranging from 100 to 1 000. This characteristic further exacerbates the inefficiencies encountered during computations.

To tackle these challenges, a ReRAM-based GCN accelerator named REFLIP[28] is proposed, encompassing three design levels. Firstly, by leveraging the crossbar structure of ReRAM, REFLIP devises a unified hardware architecture. This architecture can effectively fulfill the computational requirements of GCN aggregation and combination stages. Secondly, REFLIP employs distinctive mapping strategies to maximize efficiency. This includes utilizing a layer-wise weight mapping strategy during the combination stage to address the limitations of crossbar resources and a flipped mapping strategy during the aggregation stage to reduce the proportion of inefficient computations. Lastly, REFLIP further enhances performance through software/hardware co-optimizations. On the software level, REFLIP employs a hybrid execution model to mitigate data movement costs and enhance computational efficiency. On the hardware level, REFLIP introduces specialized hardware units that leverage GCN locality to minimize the number of conversions between digital and analog signals.

4.2.3 *Heterogeneous PIM-Based Graph Processing*

In practical scenarios, modern graph analytics workloads often exhibit irregular patterns, with most vertices having only a few edges. When these patterns are mapped onto tightly coupled ReRAM crossbars, many ReRAM cells remain unutilized as they store zero values. This results in unnecessary resistance writes and analog-signal conversions, leading to additional performance and energy costs. Unfortunately, these costs cannot be easily offset by the benefits gained from employing ReRAM.

Digital CMOS-based PIM is a promising solution to the challenges posed by analog ReRAM PIM. This technology integrates digital processing units within 3D-stacked memory's logic layer. It enables flexible computation at a finer granularity of scalar instead of the matrix level of the ReRAM crossbar. Additional-

ly, the frequency of the CMOS-based PIM is often 10x–100x higher than the ReRAM crossbar. However, the parallelism of the CMOS-based PIM is limited compared with the ReRAM crossbar, as it performs one operation at a time.

We propose Hetraph[29], a heterogeneous processing-in-memory architecture. Hetraph integrates both ReRAM-based and CMOS-based PIM units within the same logic layer of a 3D die-stacked memory device. This architecture aims to support energy-efficient and high-performance graph processing. However, the integration of two different types of PIM units causes high synchronization and communication overhead in the heterogeneous architecture. Moreover, a key problem arises in determining which PIM unit is best suited for executing each subgraph to achieve optimal efficiency. The non-deterministic nature of activation makes it challenging to identify the valid edges associated with active vertices.

To address the challenges, Hetraph provides a hardware heterogeneity-aware mechanism and a workload offloading approach. The hardware heterogeneity-aware execution model explores an optimal tradeoff between communication overheads and synchronization. In particular, this model merges intermediate results to minimize data synchronizations and reuses data to reduce communication overhead. Additionally, Hetraph identifies subgraphs with no more than one valid edge, deemed inefficient for ReRAM-based PIM. It develops a workload offloading mechanism that efficiently identifies and offloads each subgraph to the most suitable PIM to optimize efficiency.

5 Conclusions

This paper systematically describes our work in graph processing, focusing on three critical perspectives: hardware accelerator, software engine, and novel graph task/architecture.

As for the hardware accelerator design, we identified data conflicts as a significant computational bottleneck in graph processing. To address this challenge, we developed AccuGraph as a solution. We also discovered that memory bandwidth limits the performance of graph processing. To overcome this limitation, we took advantage of HBM and developed ScalaBFS2. Finally, we found another bottleneck in existing graph processing accelerators: scalability, and presented ScalaGraph to overcome it.

As for the software engine, we designed six components: HotGraph, GraphFly, CGraph, DiGraph, GraSu, and FBSGraph. These components are tailored for hardware platforms, including CPUs, GPUs, FPGAs, and distributed platforms. Each component has been carefully designed based on its hardware architecture to optimize performance and efficiency.

Finally, we explored the domains of emerging workloads and hardware architectures. We presented several innovative solutions for emerging graph workloads, including FNNG, XuLin, and MetaNMP. These approaches demonstrate our progress on emerging graph processing workloads and validate their effectiveness. For emerging hardware architectures, we developed Spara, ReFlip, Hetraph, and other notable designs. These contributions demonstrate their potential for graph processing on emerging hardware architectures.

Throughout the paper, we aimed to provide a comprehensive overview of our work and highlight our significant contributions to hardware architecture, software system design, and the development of novel applications for graph processing.

Acknowledgments We thank the following students for their contributions, where Ke-Xin Li and Bing Zhu participated in the work on the graph accelerators design, Dan Chen, Zhao-Zeng An, Yu-Jian Liao, Qian-Ge Shen, Dong-Hao He, Shi-Jun Li, Xin Ning, and Zhi-Ying Huang participated in the work on the graph software environment, and Dan Chen, Chao-Qiang Liu, Hai-Feng Liu, Hai-Heng He, and Zhao-Zeng An participated in the work on the emerging hardware/software co-design.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Wu S W, Sun F, Zhang W T, Xie X, Cui B. Graph neural networks in recommender systems: A survey. *ACM Computing Surveys*, 2023, 55(5): Article No. 97. DOI: 10.1145/3535101.

[2] Bullmore E, Sporns O. Complex brain networks: Graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 2009, 10(3): 186–198. DOI: 10.1038/NRN2575.

[3] Wang B Y, Dabbaghjamanesh M, Kavousi-Fard A, Mehraeen S. Cybersecurity enhancement of power trading within the networked microgrids based on blockchain

and directed acyclic graph approach. *IEEE Trans. Industry Applications*, 2019, 55(6): 7300–7309. DOI: 10.1109/TIA.2019.2919820.

[4] Yin J, Tang M J, Cao J L, You M S, Wang H, Alazab M. Knowledge-driven cybersecurity intelligence: Software vulnerability coexploitation behavior discovery. *IEEE Trans. Industrial Informatics*, 2023, 19(4): 5593–5601. DOI: 10.1109/TII.2022.3192027.

[5] Luo J W, He M K, Pan W K, Ming Z. BGNN: Behavior-aware graph neural network for heterogeneous session-based recommendation. *Frontiers of Computer Science*, 2023, 17(5): 175336. DOI: 10.1007/s11704-022-2100-y.

[6] He D L, Yuan P P, Jin H. Answering reachability queries with ordered label constraints over labeled graphs. *Frontiers of Computer Science*, 2024, 18(1): 181601. DOI: 10.1007/s11704-022-2368-y.

[7] Gui C Y, Zheng L, He B S, Liu C, Chen X Y, Liao X F, Jin H. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology*, 2019, 34(2): 339–371. DOI: 10.1007/S11390-019-1914-Z.

[8] Chen D, Jin H, Zheng L, Huang Y, Yao P C, Gui C Y, Wang Q G, Liu H F, He H H, Liao X F, Zheng R. A general offloading approach for near-DRAM processing-in-memory architectures. In *Proc. the 2022 IEEE International Parallel and Distributed Processing Symposium*, May 2022, pp.246-257. DOI: 10.1109/IPDPS53621.2022.00032.

[9] Yao P C, Zheng L, Liao X F, Jin H, He B S. An efficient graph accelerator with parallel data conflict management. In *Proc. the 27th International Conference on Parallel Architectures and Compilation Techniques*, Nov. 2018, Article No. 8. DOI: 10.1145/3243176.3243201.

[10] Jin H, Chen D, Zheng L, Huang Y, Yao P C, Zhao J, Liao X F, Jiang W B. Accelerating graph convolutional networks through a PIM-accelerated approach. *IEEE Trans. Computers*, 2023, 72(9): 2628–2640. DOI: 10.1109/TC.2023.3257514.

[11] Wang D W, Cui W Q. An efficient graph data compression model based on the germ quotient set structure. *Frontiers of Computer Science*, 2022, 16(6): 166617. DOI: 10.1007/s11704-022-1489-7.

[12] Fang P, Wang F, Shi Z, Feng D, Yi Q X, Xu X H, Zhang Y X. An efficient memory data organization strategy for application-characteristic graph processing. *Frontiers of Computer Science*, 2022, 16(1): Article No. 161607. DOI: 10.1007/s11704-020-0255-y.

[13] Yao P C, Zheng L, Huang Y, Wang Q G, Gui C Y, Zeng Z, Liao X F, Jin H, Xue J L. ScalaGraph: A scalable accelerator for massively parallel graph processing. In *Proc. the 2022 IEEE International Symposium on High-Performance Computer Architecture*, Apr. 2022, pp.199–212. DOI: 10.1109/HPCA53966.2022.00023.

[14] Yao P C, Zheng L, Zeng Z, Huang Y, Gui C Y, Liao X F, Jin H, Xue J L. A locality-aware energy-efficient accelerator for graph mining applications. In *Proc. the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2020, pp.895–907. DOI: 10.1109/MICRO50266.2020.00077.

[15] Rahman S, Abu-Ghazaleh N, Gupta R. GraphPulse: An event-driven hardware accelerator for asynchronous graph processing. In *Proc. the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2020, pp.908–921. DOI: 10.1109/MICRO50266.2020.00078.

[16] Jin H, Yao P C, Liao X F. Towards dataflow based graph processing. *Science China Information Sciences*, 2017, 60(12): Article No. 126102. DOI: 10.1007/s11432-017-9226-8.

[17] Li K X, Xu S X, Shao Z Y, Zheng R, Liao X F, Jin H. ScalaBFS2: A high performance BFS accelerator on an HBM-enhanced FPGA chip. *ACM Trans. Reconfigurable Technology and Systems*. DOI: 10.1145/3650037. (accepted)

[18] Zhang Y, Liao X F, Jin H, Gu L, Tan G, Zhou B B. HotGraph: Efficient asynchronous processing for real-world graphs. *IEEE Trans. Computers*, 2017, 66(5): 799–809. DOI: 10.1109/TC.2016.2624289.

[19] Chen D, Gui C Y, Zhang Y, Jin H, Zheng L, Huang Y, Liao X F. GraphFly: Efficient asynchronous streaming graphs processing via dependency-flow. In *Proc. the 2022 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2022. DOI: 10.1109/SC41404.2022.00050.

[20] Zhang Y, Liao X F, Jin H, Gu L, He L G, He B S, Liu H K. CGraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *Proc. the 2018 USENIX Annual Technical Conference*, Jul. 2018, pp.441–452. https://www.usenix.org/system/files/conference/atc18/atc18-zhang-yu.pdf, Oct. 2023.

[21] Zhang Y, Liao X F, Jin H, He B S, Liu H K, Gu L. DiGraph: An efficient path-based iterative directed graph processing system on multiple GPUs. In *Proc. the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2019, pp.601–614. DOI: 10.1145/3297858.3304029.

[22] Wang Q G, Zheng L, Huang Y, Yao P C, Gui C Y, Liao X F, Jin H, Jiang W B, Mao F B. GraSU: A fast graph update library for FPGA-based dynamic graph processing. In *Proc. the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2021, pp.149–159. DOI: 10.1145/3431920.3439288.

[23] Zhang Y, Liao X F, Jin H, Gu L, Zhou B B. FBSGraph: Accelerating asynchronous graph processing via forward and backward sweeping. *IEEE Trans. Knowledge and Data Engineering*, 2018, 30(5): 895–907. DOI: 10.1109/TKDE.2017.2781241.

[24] Liu C Q, Liu H F, Zheng L, Huang Y, Ye X Y, Liao X F, Jin H. FNNG: A high-performance FPGA-based accelerator for K-nearest neighbor graph construction. In *Proc. the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb. 2023, pp.67–77. DOI: 10.1145/3543622.3573189.

[25] Wang Q G, Zheng L, Hu A, Huang Y, Yao P C, Gui C Y, Liao X F, Jin H, Xue J L. A data-centric accelerator for high-performance hypergraph processing. In *Proc. the 55th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2022, pp.1326–1341. DOI: 10.1109/MICRO56248.2022.00088.

[26] Chen D, He H H, Jin H, Zheng L, Huang Y, Shen X Y, Liao X F. MetaNMP: Leveraging Cartesian-like product to accelerate HGNNs with near-memory processing. In *Proc. the 50th Annual International Symposium on Computer Architecture*, Jun. 2023, Article No. 56. DOI: 10.1145/3579371.3589091.

[27] Zheng L, Zhao J S, Huang Y, Wang Q G, Zeng Z, Xue J L, Liao X F, Jin H. Spara: An energy-efficient ReRAM-based accelerator for sparse graph analytics applications. In *Proc. the 2020 IEEE International Parallel and Distributed Processing Symposium*, May 2020, pp.696–707. DOI: 10.1109/IPDPS47924.2020.00077.

[28] Huang Y, Zheng L, Yao P C, Wang Q G, Liao X F, Jin H, Xue J L. Accelerating graph convolutional networks using crossbar-based processing-in-memory architectures. In *Proc. the 2022 IEEE International Symposium on High-Performance Computer Architecture*, Apr. 2022, pp.1029–1042. DOI: 10.1109/HPCA53966.2022.00079.

[29] Huang Y, Zheng L, Yao P C, Zhao J S, Liao X F, Jin H, Xue J L. A heterogeneous PIM hardware-software co-design for energy-efficient graph processing. In *Proc. the 2020 IEEE International Parallel and Distributed Processing Symposium*, May 2020, pp.684–695. DOI: 10.1109/IPDPS47924.2020.00076.

[30] Ham T J, Wu L S, Sundaram N, Satish N, Martonosi M. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proc. the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2016. DOI: 10.1109/MICRO.2016.7783759.

[31] Dai G H, Huang T H, Chi Y Z, Xu N Y, Wang Y, Yang H Z. ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture. In *Proc. the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2017, pp.217–226. DOI: 10.1145/3020078.3021739.

[32] Chen X Y, Chen Y, Cheng F, Tan H S, He B S, Wong W F. ReGraph: Scaling graph processing on HBM-enabled FPGAs with heterogeneous pipelines. In *Proc. the 55th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2022, pp.1342–1358. DOI: 10.1109/MICRO56248.2022.00092.

[33] Yan M Y, Hu X, Li S C, Basak A, Li H, Ma X, Akgun I, Feng Y J, Gu P, Deng L, Ye X C, Zhang Z M, Fan D R, Xie Y. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In *Proc. the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2019, pp.615–628. DOI: 10.1145/3352460.3358318.

[34] Zhang Y F, Gao Q X, Gao L X, Wang C R. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Trans. Parallel and Distributed Systems*, 2014, 25(8): 2091–2100. DOI: 10.1109/TPDS.2013.235.

[35] Gonzalez J E, Low Y, Gu H J, Bickson D, Guestrin C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. the 10th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2012, pp.17–30.https://www.usenix.org/system/files/conference/osdi12/osdi12-final-167.pdf, Oct. 2023.

[36] Vora K, Gupta R, Xu G Q. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proc. the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2017, pp.237–251. DOI: 10.1145/3037697.3037748.

[37] Mariappan M, Vora K. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In *Proc. the 14th EuroSys Conference*, Mar. 2019, Article No. 25. DOI: 10.1145/3302424.3303974.

[38] Wang Y Z H, Davidson A, Pan Y C, Wu Y D, Riffel A, Owens J D. Gunrock: A high-performance graph processing library on the GPU. In *Proc. the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2016, Article No. 11. DOI: 10.1145/2851141.2851145.

[39] Ben-Nun T, Sutton M, Pai S, Pingali K. Groute: An asynchronous multi-GPU programming model for irregular computations. *ACM SIGPLAN Notices*, 2017, 52(8): 235–248. DOI: 10.1145/3155284.3018756.

[40] Dong W, Moses C, Li K. Efficient *k*-nearest neighbor graph construction for generic similarity measures. In *Proc. the 20th International Conference on World Wide Web*, Mar. 2011, pp.577–586. DOI: 10.1145/1963405.1963487.

[41] Wang Q G, Zheng L, Yuan J R, Huang Y, Yao P C, Gui C Y, Hu A, Liao X F, Jin H. Hardware-accelerated hypergraph processing with chain-driven scheduling. In *Proc. the 2022 IEEE International Symposium on High-Performance Computer Architecture*, Apr. 2022, pp.184–198. DOI: 10.1109/HPCA53966.2022.00022.

[42] Hu M, Strachan J P, Li Z Y, Grafals E M, Davila N, Graves C, Lam S, Ge N, Yang J J, Williams R S. Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication. In *Proc. the 53rd Annual Design Automation Conference*, Jun. 2016, Article No. 19. DOI: 10.1145/2897937.2898010.

[43] Chi P, Li S C, Xu C, Zhang T, Zhao J S, Liu Y P, Wang Y, Xie Y. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *Proc. the 43rd Annual International Symposium on Computer Architecture*, Jun. 2016, pp.27–39. DOI: 10.1109/ISCA.2016.13.

[44] Song L H, Zhuo Y W, Qian X H, Li H, Chen Y R. GraphR: Accelerating graph processing using ReRAM. In *Proc. the 2018 IEEE International Symposium on High Performance Computer Architecture*, Feb. 2018, pp.531–543. DOI: 10.1109/HPCA.2018.00052.

[45] Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks. arXiv: 1609.02907, 2016. https://arxiv.org/abs/1609.02907, Mar. 2024.

[46] Jin T S, Dai H Q, Cao L J, Zhang B C, Huang F Y, Gao Y, Ji R R. Deepwalk-aware graph convolutional networks. *Science China Information Sciences*, 2022, 65(5): 152104. DOI: 10.1007/s11432-020-3318-5.

[47] Bai J Y, Guo J, Wang C C, Chen Z Y, He Z, Yang S, Yu P P, Zhang Y, Guo Y W. Deep graph learning for spatially-varying indoor lighting prediction. *Science China Infor-*

mation Sciences, 2023, 66(3): Article No. 132106. DOI: 10.1007/s11432-022-3576-9.

[48] Fey M, Lenssen J E. Fast graph representation learning with PyTorch geometric. arXiv: 1903.02428, 2019. https://arxiv.org/abs/1903.02428, Mar. 2024.

Xiao-Fei Liao is a professor in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan. He received his Ph.D. degree in computer science and engineering from HUST, Wuhan, in 2005. He was awarded Excellent Youth Award from the National Science Foundation of China in 2018, and CCF-IEEE CS Young Computer Scientist Award in 2017. His research interests are in the areas of computer architecture, system software, and big data processing.

Wen-Ju Zhao is a Ph.D. candidate in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan. His research interests include computer architecture and graph neural networks.

Hai Jin is a 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, Straβe der Nationen. Jin worked at The University of Hong Kong, Hong Kong, between 1998 and 2000, and as a visiting scholar at the University of Southern California, Los Angeles, between 1999 and 2000. He was awarded Excellent Youth Award from the National Natural Science Foundation of China in 2001. He has co-authored 22 books and published over 900 research papers. His research interests include computer architecture, virtualization technology, distributed computing, big data processing, network storage, and network security.

Peng-Cheng Yao received his Ph.D. degree in computer science and technology from the Huazhong University of Science and Technology, Wuhan, in 2022. He is now a postdoctoral fellow at Zhejiang Lab, Hangzhou. His research interests include graph processing and domain specific accelerator.

Yu Huang received his Ph.D. degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, in 2022. He is now a postdoctoral fellow at Zhejiang Lab, Hangzhou. His research focuses on computer architecture, graph processing, and processing in memory.

Qing-Gang Wang received his Ph.D. degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, in 2023. He is now a postdoctoral fellow at Zhejiang Lab, Hangzhou. His current research interests include graph processing and reconfigurable computing.

Jin Zhao received his Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, in 2022. He is now a postdoctoral fellow at Zhejiang Lab, Hangzhou. His research focuses on computer architecture, system software, runtime optimization, programming model, and big data processing.

Long Zheng received his Ph.D. degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, in 2016. He is currently an associate professor with the School of Computer Science and Technology, HUST, Wuhan. His current research interests include program analysis, runtime systems, and heterogeneous computing with a particular focus on graph processing.

Yu Zhang received his Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, in 2016. His research focuses on computer architecture and system, runtime optimization, programming model, and big data processing.

Zhi-Yuan Shao received his Ph.D. degree in computer science and technology from Huazhong University of Science and Technology (HUST), Wuhan, in 2005. He is now a professor of computer science and engineering at HUST in Wuhan. His research interests are in the areas of graph computing, computing system, and big-data processing.