2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2027-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPCA53966.2022.00023

ScalaGraph: A Scalable Accelerator for Massively Parallel Graph Processing

Pengcheng Yao†, Long Zheng†, Yu Huang†, Qinggang Wang†, Chuangyi Gui†, Zhen Zeng†,
Xiaofei Liao†, Hai Jin†, Jingling Xue‡
†*National Engineering Research Center for Big Data Technology and System/Services Computing Technology and System Lab/Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China*
‡*School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia*
{*pcyao, longzh, yuh, qgwang, chygui, zzeng, xfliao, hjin*}*@hust.edu.cn, jingling@cse.unsw.edu.au*

***Abstract*—Graph processing is promising to extract valuable insights in graphs. Nowadays, emerging 3D-stacked memories and silicon technologies can provide over terabytes per second memory bandwidth and thousands of *processing elements* (PEs) to meet the high hardware demand of graph applications. However, this leap in hardware capability does not result in a huge increase but even a degradation sometimes in performance for graph processing. In this paper, we discover that the *centralized* on-chip memory hierarchy adopted in existing graph accelerators is the villain causing poor scalability due to its quadratic increase of hardware overheads with respect to the number of PEs.**

We present a novel *distributed* on-chip memory hierarchy by leveraging the *network-on-chip* (NoC) to enable massively parallel graph processing. We architect ScalaGraph, a brand new graph processing accelerator, to exploit this insight. ScalaGraph adopts a software-hardware co-design to minimize NoC communication overheads via an efficient row-oriented dataflow mapping and runtime aggregation. A specialized scheduling mechanism is also proposed to improve load imbalance. Our results on a Xilinx Alveo U280 FPGA card show that ScalaGraph on a modest configuration of 512 PEs achieves 2.2× and 3.2× speedups over a state-of-the-art graph accelerator GraphDyns and a GPU-based graph system Gunrock, respectively. Moreover, ScalaGraph enables supporting at least 1,024 PEs with nearly linear performance scaling while GraphDyns fails to work.

***Keywords*-graph processing; accelerator; scalability**

I. Introduction

Graphs are widely used to represent complex relationships between entities. By iteratively traversing graph data, graph processing can mine valuable insights in many fields, including community detection [1], advertisement recommendation [2], and bioinformatics [3]. With the ever-increasing data volume, a lot of specialized algorithms [4]–[7] and systems [8]–[13] have been developed to process large-scale graphs efficiently. However, the performance of graph processing is still limited to the underlying general-purpose architectures [14]–[16]. Real-world graphs are often extremely sparse, where each vertex connects with a small portion of its graph. For example, the Twitter graph [1] has 41 million vertices, but each vertex only connects with 35 neighbors on average. The unpredictable traversal of sparse edges introduces a large number of random memory accesses, resulting in expensive off-chip communications in two aspects at least. First, memory access latency increases severely due to frequent cache misses [17]. Second, only a portion of a 64-byte cacheline (e.g., 4 bytes) is used, which gravely wastes off-chip memory bandwidth [18].

Therefore, general-purpose architectures are not an ideal platform for graph processing. In the past few years, there has been a significant interest in developing dedicated graph accelerators to improve memory inefficiencies with sophisticated on-chip memory management [18]–[23]. GraphPulse [24] implements a large event queue to store all in-flight vertex updates on-chip and therefore eliminates most of the random vertex accesses. GraphDyns [21] improves the on-chip data utilization by applying explicit prefetching and vectorized data access. These earlier advances on graph accelerators have successfully delivered nearly tenfold performance gains over general-purpose processors, but little thought has been given to the scalability issue of their on-chip memory hierarchy itself.

Emerging 3D-stacked memories, e.g., *high-bandwidth memory* (HBM), can deliver up to *terabytes-per-second* (TB/s) memory bandwidth [25]. This significant boost in memory efficiency exposes a great opportunity for developing a high-throughput graph accelerator. We assume an accelerator running on an FPGA platform, which is widely used in specializing graph processing architectures due to its high performance and flexibility [19], [20], [22]. The accelerator is well-designed to access all graph data sequentially, with each edge represented in 4 bytes. Under a running frequency of 250MHz, the accelerator is expected to simultaneously process 1,024 ($\frac{1024}{0.25\times4}$) edges to make full use of 1TB/s memory bandwidth, with at least 1,024 *processing elements* (PEs) required to meet the above requirement.

Of course, the emerging silicon technology is capable of supporting thousands of PEs easily [26]. However, the performance of graph processing applications does not make a huge increase due to the significant leap of the underlying hardware capability (as discussed in Section II-B). The villain causing this issue mainly lies in a *centralized* on-chip memory hierarchy adopted in existing graph accelerators [18]–[21], [24], in the sense that the on-chip memory is shared by all PEs in a centralized manner. Specifically, each PE is connected with all memory partitions through a *crossbar*-like switch where all pairwise input-output ports allow direct communications, therefore

978-1-6654-2027-3/22/$31.00 ©2022 IEEE
DOI 10.1109/HPCA53966.2022.00023

achieving minimized on-chip memory access latency and avoiding inter-PE pipeline stalls.

However, the required resources and hardware complexity of this centralized mechanism increase exponentially with respect to the number of PEs, making it difficult to deploy massive PEs efficiently and effectively. In this context, the complex crossbar interconnection will quickly become a performance bottleneck when the number of PEs is large, leading to poor scalability. For example, from our hands-on experiments on FPGA, existing graph accelerators generally suffer from severe reduction in frequency and performance when the number of PEs exceeds 128. Worse still, if the number of PEs exceeds 256, the crossbar would cause the route failure of the accelerator. In this paper, we focus on specializing an improved on-chip memory hierarchy to fill the gap between high bandwidth memory and massive PEs such that their hardware potential can be fully exploited.

To achieve the above goal, rather than focusing primarily on minimizing memory overheads [18], [21], it is of great necessity to consider both hardware complexity and memory efficiency in finding an appropriate tradeoff between them. In other words, we argue that a high throughput graph accelerator should *build scalability on top of efficiency* to meet the ever-increasing demand of massive parallelism and high memory bandwidth.

In this paper, we present ScalaGraph, a scalable graph accelerator, which is built upon a **distributed on-chip memory hierarchy**. The on-chip memory in ScalaGraph is divided into many slices with each associated with a PE. All these slices are connected via a mesh-like *network-on-chip* (NoC). Unlike the full connection adopted in previous works [20], [21], [27], each PE in ScalaGraph only needs to communicate with its neighbors. Since the number of neighbors for a PE is very small, a high scalability can be preserved. To minimize the NoC communication overhead, we propose a software-hardware co-design. In software, we design a new architecture-specific algorithm mapping to map the graph workloads to the accelerator with minimized communications. In hardware, we architect the pipelined data aggregation to coalesce vertex updates for reducing NoC routing conflicts at runtime. In addition, we also present a *degree-aware scheduling* mechanism to achieve load-balancing in scheduling high- and low-degree vertices, and an *inter-phase pipelining* mechanism to accelerate the inter-iteration process efficiently.

ScalaGraph aims to improve on-chip memory scalability, which is orthogonal to the past work on developing existing graph processing accelerators for improving off-chip memory efficiency by using, e.g., efficient batching [28], caching [18], [24], [29], and prefetching [21], [30].

In summary, we make the following contributions:

- We revisit existing graph processing accelerators on emerging high bandwidth memory with massive parallelism, and identify their centralized on-chip memory hierarchy as the main limiting factor for scalability.
- We architect ScalaGraph, a scalable graph accelerator with a distributed on-chip memory hierarchy that achieves high scalability and software-hardware co-designs for boosting performance at a low cost.
- We implement ScalaGraph in RTL and evaluate it on a Xilinx Alveo U280 accelerator card. Our results show that ScalaGraph enables scaling with thousands of PEs and achieves 2.2× and 3.2× performance improvement over GraphDyns [21] and Gunrock [31], respectively.

Vertex-Centric Programming Model

Input: V_{active} – Active vertex set
Input: V_{prop} – Vertex Property Array
Output: V'_{active} – Output active vertex set

```
▷ Scatter Phase
foreach vertex v ∈ V_active do
    for each edge E(v,u) of active vertex v do
        ScatterRes = Process(EWeight, V_prop[v]);
        V_temp[u] = Reduce(V_temp[u], ScatterRes) ;
    end
end
▷ Apply Phase
foreach vertex v do
    ApplyRes = Apply(V_prop[v], V_temp[v], V_const);
    if ApplyRes! = V_prop[v] then
        V_prop[v] = ApplyRes;
        V'_active.push(u);
    end
end
```

Figure 1. Pseudocode for vertex-centric programming model

The rest of this paper is organized as follows. Section II introduces the background and motivation. Section III presents ScalaGraph. Section IV introduces the software-hardware co-designs. Section V describes and analyzes the results. We survey the related work in Section VI and conclude the paper in Section VII.

II. Background And Motivation

In this section, we first review graph processing and its hardware accelerators, then examine the limitations of existing accelerators, and finally, motivate our approach.

A. Irregularities in Graph Processing

The *vertex-centric model* (VCM) is widely used to write parallel graph applications [8] easily. In the VCM model, each vertex is considered as an individual processing element and communicates with its neighbors through associated edges. Figure 1 shows a typical graph processing procedure written in VCM. There are two main phases: *Scatter* and *Apply*. In the Scatter phase, all edges of each active vertex are traversed to update the temporary vertex properties V_{temp} of its neighbors. In the Apply phase, each vertex updates its property using V_{temp} and a constant value V_{const}. Finally, the updated vertices will form a new active vertex set as the input of the next iteration. `Process`, `Reduce`, and `Apply` are user-defined to express different graph algorithms.

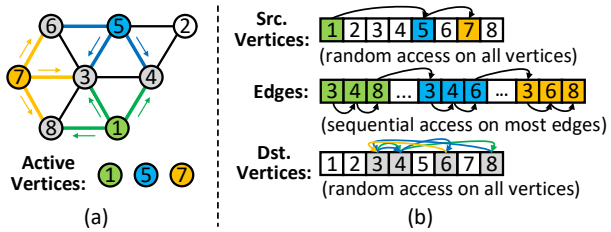


Figure 2. An example for illustrating memory accesses in graph processing, including (a) input graph and active vertices, and (b) memory access patterns of vertices and edges

VCM exposes parallelism in graph processing, but its performance is limited due to the well-known irregularity challenge in graph processing. The graph irregularity can induce a large number of random accesses in the Scatter phase (Lines 4 and 5). Figure 2 exemplifies memory access patterns in graph processing for a specific iteration, where ❶, ❺, and ❼ indicate active vertices to be processed. As is observed, these active vertices and their neighbors have inconsecutive indices. In the Scatter phase, the memory accesses to them are therefore completely random. For example, when traversing the neighbors of ❶, a random access is made between ❹ and ❽. Moreover, traversing ❺ will generate another random access between ❶ and ❺.

These random memory accesses lead to server performance-limiting issues when running graph algorithms on general-purpose architectures. First, random memory accesses can significantly increase the memory latency, resulting in inefficient data movement between processors and off-chip memory. For example, the hit ratio of L2 cache is only 10% for graph workloads on CPU, rendering most of the execution time to be spent on memory accesses [17]. Second, they will cause only a small portion of a fetched 64-byte cacheline to be used, increasing further the total number of memory requests. An extreme example is that 129× more off-chip memory accesses may incur against the ideal situation (with all data reused) [18].

B. Pitfalls of Existing Graph Accelerators

Figure 3 shows a typical architecture template for existing graph accelerators [18]–[21], [24], [27], [32], consisting of a scheduler, a processor, and an on-chip memory. The scheduler and processor include several *scheduler elements* (SEs) and PEs, respectively. Due to unpredictable edge traversal patterns in graph processing, each on-chip *memory partition* (MP) is often fully connected to all PEs in a centralized fashion. When an edge is received from a SE, the PE will shuffle it to the MP storing the associated vertex data through a crossbar-like switch. Therefore, the edges updating the same vertex will be shuffled to the same MP and processed sequentially without data conflicts, therefore avoiding inter-pipeline stalls as well. By considering together the centralized on-chip memory hierarchy and specialized memory optimizations [18], [21], [24] used, graph accelerators achieve impressive results over conventional architectures.

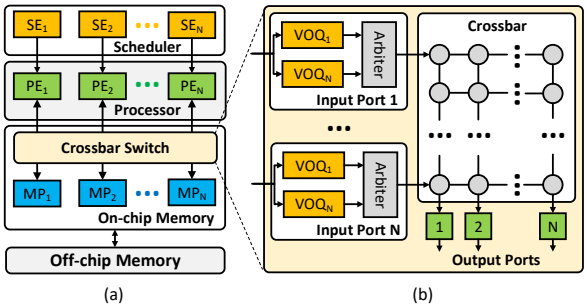
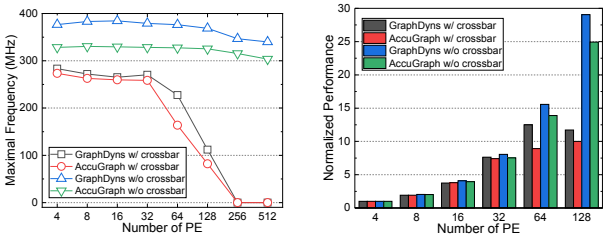


Figure 3. A typical graph processing accelerator, including (a) overall architecture and (b) crossbar switch

Limitation Analysis. While the centralized crossbar preserves high efficiency, little thought is given to its scalability. Compared to traditional DRAM, emerging 3D-stacked memory delivers at least one order of magnitude higher memory bandwidth. For example, NVIDIA A100 provides up to 2TB/s HBM bandwidth [25], while the representative DDR4 2400 has only 19.2GB/s. This significant improvement in memory bandwidth offers a great chance for developing a high throughput graph accelerator. In graph processing, the edge data is often hardly reused since a directional edge can be traversed only once at each iteration. Therefore, each edge, when processed, must occupy a unique PE in each cycle. As mentioned earlier, an accelerator running at 250MHz must use at least 1,024 PEs to process 1,024 edges simultaneously to utilize 1TB/s bandwidth.

However, when the number of PEs increases, the hardware overhead of the centralized crossbar increases significantly and becomes a major bottleneck quickly. Figure 2(b) shows a typical crossbar switch with the *virtual output queue* (VOQ). It is clear that the hardware complexity of both crossbar connection and arbiter in VOQ are O(N^2). That is, their hardware overheads grow quadratically to N, i.e., the number of inputs/outputs. When the number of PEs increases, the increased overheads can significantly increase the underlying hardware latency, thus leading to severe performance degradation. Moreover, the computation logics in graph processing are as simple as arithmetic operators (e.g., +, -, and ×), causing the on-chip memory to usually take more than 85% of the overall chip area [21]. Building a complex connection between the small-area computation units and large-area on-chip memory partitions can increase the latency further. This problem is particularly serious for the FPGA platform, since its hardware resources are often pre-allocated in the physical chip, making it sensitive to the hardware complexity of an accelerator design.

We further conduct a set of experiments to investigate the performance impact arising from the centralized on-chip memory, by benchmarking state-of-the-art FPGA-based accelerator AccuGraph [20] and ASIC-based accelerator GraphDyns [21] on a Xilinx Avleo U280 FPGA card which provides 460GB/s memory bandwidth. Since the two graph accelerators are not publicly available, we prototype their



(a) Maximal Frequency (b) Normalized Performance

Figure 4. The effect of the crossbar switch in terms of (a) maximal frequency and (b) performance. All results are normalized to a baseline implementation using 4 PEs.

Table I
GRAPH DATASETS

Graph	Flickr	Pokec	LiveJournal	Orkut
#Vertices	0.82M	1.63M	4.84M	3.07M
#Edges	9.84M	30.62M	68.99M	234.37M

architectures according to their papers. Each accelerator is configured with a 4MB on-chip memory using BRAM resources. We perform one iteration of PageRank to characterize the maximal throughput of an accelerator [8]. Table I shows the graph datasets used. For comparison purposes, we also compare each accelerator against a version by removing the crossbar without ensuring accuracy. Since all edges are processed in each iteration in PageRank, the amount of computation performed will stay the same even if the crossbar is removed, therefore providing a precise estimation. For crossbar, we adopt the widely-used swizzle-switch crossbar implementation [33], [34].

Figure 4 shows the results. The results with less than 4 PEs are excluded since the computation component area is too small to influence the frequency significantly. As shown in Figure 4(a), the two accelerators carrying the crossbar suffer from significant frequency reduction. When the number of PEs exceeds 64, their running frequencies drop severely from 300MHz to 100MHz. Scaling to 256 or 512 PEs can even lead to route failures in the synthesis tool due to high hardware complexity. As a result, the performance scalability of two graph accelerators suffers significantly when the number of PEs increases to 128.

Figure 4(b) shows the performance results. When scaling from 4 to 64 PEs, the two graph accelerators can obtain 10∼12× speedups, achieving almost ideal scalability (16×). However, when scaling from 64 to 128 PEs, their performance improves slightly or even drops due to severe frequency reduction. The results for 256 PEs and 512 PEs are not included since the synthesis tool fails to find a valid routing solution. In contrast, the two accelerators without the crossbar can maintain a high running frequency of 300MHz, achieving almost linear performance scalability.

C. Distributed On-chip Memory

As shown from the experimental results in Figure 4(a), the off-chip memory bandwidth is far from saturation, which can be as low as no more than 12% ($\frac{4\times0.112\times128}{460}$) with a running

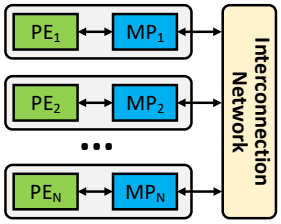


Figure 5. An example of distributed memory model

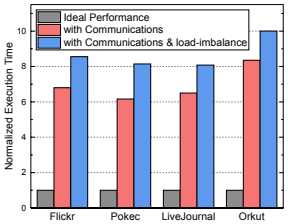


Figure 6. Overheads arising in implementing a Mesh network

frequency of 0.112GHz and 4-byte edges when the number of PEs increases to 128. On-chip memory scalability, rather than off-chip communication, becomes the most primitive villain that limits the performance of graph accelerators in the case of massive computation parallelism. We argue that the above side-effects can be avoided when designing a high-throughput graph accelerator by building scalability on top of efficiency. A scalable graph accelerator design should consider both hardware overhead and memory efficiency.

To this end, we propose to use a *distributed* on-chip memory hierarchy, in which the MPs are decentralized and associated directly with the PE. This design is inspired from the distributed memory model [35] widely adopted in multiprocessors and distributed computing. Figure 5 shows a typical distributed memory model consisting of an array of computing nodes connected via a simple interconnection, e.g., mesh-like network. It can achieve high scalability by using direct communications between computing nodes. We apply this idea to the on-chip memory design of graph accelerators. Instead of scheduling edges with a centralized shuffling mechanism, distributed on-chip memory associates the MPs with PEs directly and uses a simple NoC to connect them. When an edge is received from the off-chip memory, it will be sent to a PE storing the connected vertex via network routing. With each PE connected to a limited number of neighbors, the hardware complexity is reduced significantly.

Challenges. However, realizing the distributed on-chip memory in a graph accelerator is challenging. One challenge is to reduce the increased on-chip communication overheads caused by network routing. Unlike the crossbar with a direct connection between all inputs and outputs (Figure 3), typical NoCs often need to route the workload several times for a remote PE access through internal routing links, resulting in increased on-chip communications (i.e., the total amount of traffic injected into the on-chip network). Moreover, two workloads sent from different PEs may share the same routing link. This will result in annoying routing conflicts. As shown in Figure 6, the increased on-chip communications can lead to 6.9× performance slowdown on average by running PageRank on a 16 × 16 Mesh network.

Another challenge is to handle load imbalance caused by the power-law edge distribution of real-world graphs, where a few vertices connect with most edges. In other words, a PE storing a high-degree vertex may generate more workloads

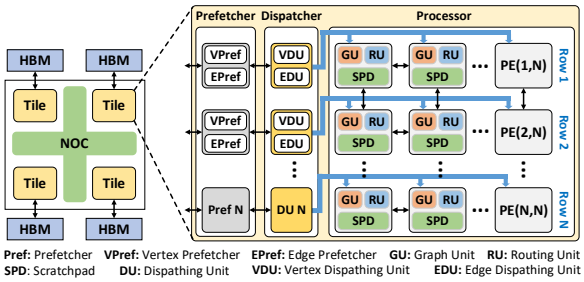


Figure 7. ScalaGraph architecture

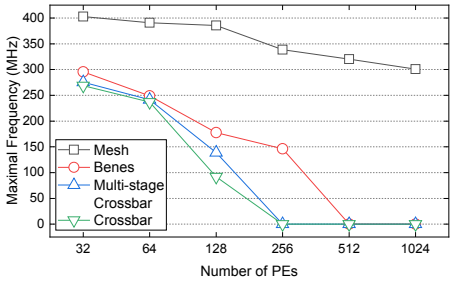


Figure 8. Frequency of different NoCs for different number of PEs

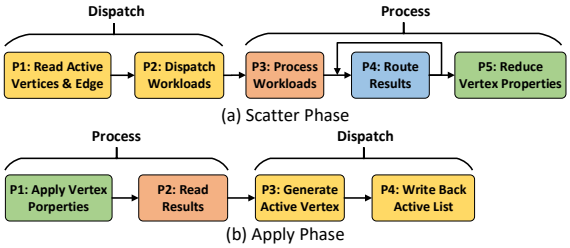


Figure 9. ScalaGraph workflow

than a PE storing a low-degree one, leading to significant load imbalance. As shown in Figure 6, the execution time will be degraded by 1.74× further.

III. Overview

In this section, we first introduce the architecture of ScalaGraph and then present its workflow.

A. Architecture

Figure 7 shows the overall architecture of ScalaGraph, which consists of several tiles connected through a NoC. Each tile processes disjoint graph partitions in its private HBM stack for scalability considerations. The memory addresses and properties of edges and active vertex list are stored in the HBM, while the vertex properties are stored on-chip. Each tile in ScalaGraph consists of four components: *prefetcher*, *dispatcher*, *processor*, and *NoC*.

Prefetcher. This module performs explicit prefetching [24] to access graph data from the off-chip HBM. Each *prefetcher* (Pref) in the prefetcher module connects to a pseudo channel of HBM to achieve high memory-level parallelism. It prefetches active vertices (including vertex IDs and edge memory addresses) and their associated edges from the HBM via a *vertex prefetcher* (Vpref) and *edge prefetcher* (Epref), respectively.

Dispatcher. Receiving graph data from the prefetcher module, this module dispatches the workloads to be processed to the *processor* module described later. The dispatcher module consists 16 *dispatching units* (DUs). Each dispatches the 64-byte active vertices and edges to a corresponding row of PEs (in Figure 7) via *vertex DU* (VDU) and *edge DU* (EDU), respectively. To achieve a load balance process, we design a degree-aware scheduling mechanism to evenly dispatch workloads to all PEs and an inter-phase pipelining mechanism to avoid PE from starvation.

Processor. This module includes the key design of distributed on-chip memory with a 16×16 PE matrix. Each PE consists of a *graph unit* (GU), a *routing unit* (RU), and a *scratchpad memory* (SPD). Each GU processes the workloads from the dispatcher module to generate vertex updates. RUs shuffle the results of GUs to the SPDs that store the corresponding destination vertices. The vertex properties are evenly partitioned to all SPDs via a simple hashing upon vertex IDs. To process a large graph whose vertex properties cannot reside in the SPDs entirely, ScalaGraph slices a graph as in Graphicionado [18], and processes all partitions in a round-robin manner. To improve NoC efficiency, we present a software-hardware co-design with a row-oriented mapping to minimize on-chip communications and an aggregation pipeline to reduce routing conflicts.

NoC. All PEs are connected through a NoC. We use 2D-Mesh as a base NoC of ScalaGraph for the following reasons. First, compared to other NoCs, Mesh has a low hardware complexity, representing a good solution for solving the scalability problem in graph processing. Figure 8 compares four representative NoCs with different hardware complexities. We select the crossbar in Figure 3 with $O(N^2)$ as a baseline. Benes [36], [37] is used as a representative NoC with $O(N \log N)$. Mesh is with $O(N)$. In addition, we also evaluate a multi-stage crossbar with several PEs multiplexed into one crossbar port [24], [32]. As shown in Figure 8, Benes and multi-stage crossbar, which achieve better scalability than the crossbar though, still suffer from significant frequency reduction or even fail to compile in case of 512 PEs. Similar results are also observed in ASIC designs, where the frequency of Benes quickly reduces from 1.5 GHz under 16 PEs to 0.6 GHz under 64 PEs [38]. In contrast, Mesh can support 1,024 PEs easily with almost negligible frequency loss.

Second, ScalaGraph focuses on addressing the common challenges faced when applying a distributed NoC to graph processing. In fact, the design of ScalaGraph is fully compatible with that of other NoCs via minor modifications. As for the problem of determining or even designing the most appropriate NoC, we leave it as an interesting future work.

B. Workflow

For generality, ScalaGraph follows the programming model in Figure 1 to write graph applications. Users do not

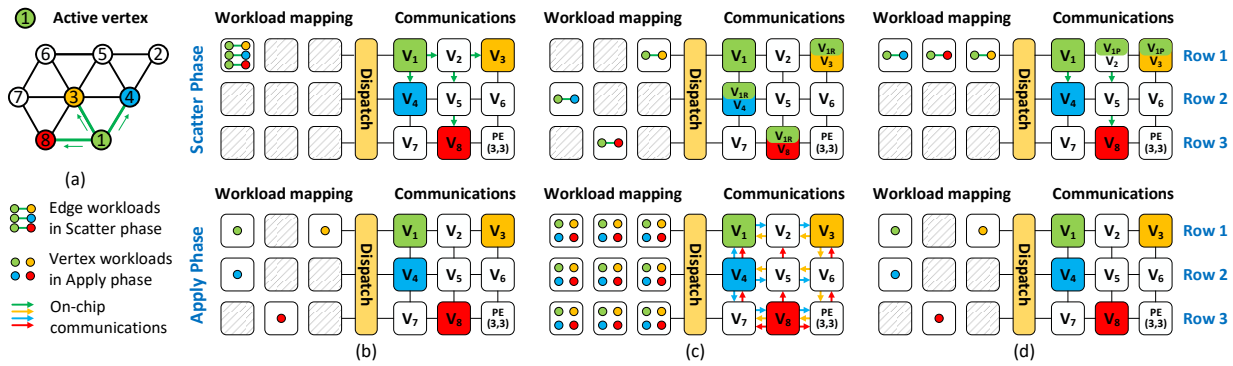


Figure 10. An illustrative example for different mappings in the Scatter and the Apply phases. (a) An input graph, (b) Source-oriented mapping, (c) Destination-oriented mapping, and (d) Our row-oriented mapping

need to understand the underlying distributed communication primitives that are transparently handled by ScalaGraph. The *compressed sparse row* (CSR) format is used for space-saving. Figure 9 further shows the workflow of running a graph application under ScalaGraph.

Scatter Workflow. First, DU reads an active vertex and its associated edges from the prefetcher in stage P1 and then dispatches workloads to the corresponding PE in stage P2. Next, the GU executes the *Process* function in Figure 1 to process workload in stage P3, and sends the results to the RU in the same PE. For the results whose destination vertices are stored locally, RU routes it to the local SPD which executes the *Reduce* function, and stores the result in stage P5. For the other results, RU routes them to the nearby RUs as a springboard for the further routing to the related PE.

Apply Workflow. First, SPD executes the *Apply* function for each vertex stored, and sends the results to GU in stage P1. Second, GU compares the results with the old vertex property, and sends the update status to DU in stage P2. DU removes the vertices whose properties are not updated in stage P3, and writes back the active vertex list to the HBM in stage P4.

IV. The ScalaGraph Design

In this section, we describe the software-hardware co-designs in ScalaGraph. We first introduce an architecture-aware data mapping and an update aggregation design that minimize the NoC communication overhead cooperatively, and then present two data-aware designs that improve load imbalance effectively in the Scatter and Apply phases.

A. Minimizing NoC Communications

In ScalaGraph, vertex properties are separately stored in different PEs. When a vertex accesses another vertex stored in a remote PE, the workload will be routed by several different RUs, significantly increasing on-chip communications. Therefore, the key to reducing on-chip communications lies in how to map a graph workload to different PEs with minimized routing distances between connected vertices. There are two types of workload-PE mapping mechanisms adopted in existing graph accelerators: *source-oriented mapping* and *destination-oriented mapping*. We next describe how they are deficient in the NoC-specific context.

Source-Oriented Mapping. Source-oriented mapping, widely adopted in graph accelerators [18], [20], [21], simply dispatches all workloads of a source vertex to a PE. Figure 10(b) shows an example of source-oriented mapping. Initially, ❶ is active to update its neighbors ❸, ❹, and ❽. Vertex properties are evenly stored to all PEs in a round-robin manner. In Scatter phase, the PE in green will store the property of ❶ and further receive its three edges. Since the destination vertices (e.g., ❸) are not stored locally, the three associated edges will be routed to the remote PEs storing the related data. In Apply phase, each PE updates the properties of its local vertices (e.g., green PE updates ❶) to generate a new list of active vertices for the next iteration.

Since all source vertex properties are stored locally, the source-oriented mapping minimizes on-chip communications for the Apply phase, but it is costly for the Scatter phase because of remote accesses to the destination vertices in edge workloads. Given K PEs, the average amount of communications for each edge workload will be $O(\sqrt{K})$. Therefore, the total per-iteration on-chip communications can be estimated to be $O(M\sqrt{K})$, where M denotes the number of edges.

Destination-Oriented Mapping. Destination-oriented mapping, adopted in some HMC-based graph accelerators [39], [40], dispatches all edge workloads with the same destination vertex to a PE that has a replica of all source vertices [19]. When a destination vertex receives an edge workload, it can directly access the replica of its source vertex locally, therefore reducing the number of communications significantly.

As shown in Figure 10(c), the edges of ❶ are sliced into several partitions based on their destination vertices. Each partition maintains an independent CSR storage and replica vertices. In Scatter phase, each PE is allocated with the workloads whose destination vertices are stored locally. These workloads can access source and destination vertices locally, thus avoiding communications. However, in Apply phase, each newly-active vertex has to update its replicas in

Table II
DIFFERENCES OF THE THREE MAPPINGS IN TERMS OF COMMUNICATION VOLUME. K DENOTES THE NUMBER OF PES. N IS THE NUMBER OF ACTIVE VERTICES, AND M DENOTES THE NUMBER OF ACTIVE EDGES.

Mechanism	Scatter	Apply	Off-chip
Source-oriented	$O(M\sqrt{K})$	$O(N)$	$O(N+M)$
Destination-oriented	0	$O(NK)$	$O(NK+M)$
Row-oriented	$O(M\sqrt{K}/2)$	$O(N)$	$O(N+M)$

all PEs. For example, ❸ needs to access the blue PE storing the edge between ❸ and ❹.

As discussed above, the destination-oriented mapping achieves minimal on-chip communications for the Scatter phase but incurs a significant amount of on-chip communications for the Apply phase, in the order of at least $O(NK)$ with N vertices. In addition, the amount of off-chip communications can reach $O(NK)$ since each partition maintains a local CSR storage. When K (the number of PEs) is large, the amount of communication incurred may exceed that incurred by the source-oriented mapping. Worse still, it needs $O(NK)$ extra storage, which can be too large to store all vertex replicas on-chip.

Row-Oriented Mapping. To make the best of both worlds, we propose a novel row-oriented mapping. Consider the matrix layout of PEs in Figure 7. The key idea is to place an edge workload in a PE, where the source (destination) vertex of this edge is located in the same row (column) as the PE. This mapping allows only inter-PE communications in the same column while the inter-PE ones in the same row are transformed into fast local accesses.

Figure 10(d) shows an example of row-oriented mapping. In Scatter phase, since ❶ is stored in the first PE of Row 1, the DU first sends the property of vertex ❶ (i.e., V_{1P}) to all PEs in Row 1. Then, it dispatches the edge workload (❶, ❽) to the second PE in Row 1 since its destination vertex ❽ is stored in a PE invovled in the same column (i.e., the PE in red). In Apply phase, our mapping proceeds similarly as the source-oriented mapping by mapping vertex workloads to a PE where their properties are stored.

Compared to the source-oriented mapping, the row-oriented mapping does not have inter-PE communications in the same row, reducing half of the on-chip communications in the Scatter phase. Compared to the destination-oriented mapping, the row-oriented mapping significantly reduces the on-chip communications in the Apply phase, and also achieves minimal off-chip communications since it only maintains a global CSR for all vertices. Table II shows the amount of communication incurred by the three mappings. We see that the row-oriented mapping has the smallest communication traffic for both Scatter and Apply phase, yielding the least communication traffic in total.

B. Reducing NoC Routing Conflicts

The row-oriented mapping achieves minimum communication traffic, but does not ensure the optimal communication

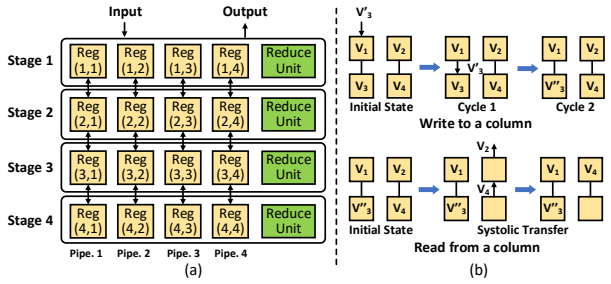


Figure 11. An example aggregation pipeline with (a) overall architecture, and (b) an example workflow for a write and a read

efficiency due to routing conflicts incurred in the same column. In the context of graph processing, it is fortunate that we have lots of opportunities to reduce these routing conflicts: (1) Routing conflicts occur only when the *Reduce* function is invoked simultaneously; (2) Further, the *Reduce* for most graph algorithms involve only simple arithmetical operations that can be often accumulated (e.g., compare and swap used in BFS) [20]; and (3) The workloads updating the same vertex are dispatched on the same column of PEs, increasing the accumulation chance further.

We further propose an aggregation pipeline. The key idea is to pre-execute the *Reduce* when a PE receives two workloads updating the same vertex in the routing time. A straightforward implementation is to compare the workload received in the current cycle with all workloads stored in the buffer, execute the Reduce function, and write the results back to the buffer. However, this incurs high overheads because all data in the buffer will be accessed in each cycle.

We therefore present a four-stage aggregation pipeline in the RU of every PE. As shown in Figure 11(a), each pipeline stage contains four registers (to store update workloads) and a reduce unit shared by them. In each cycle, once one of the four registers in the first stage receives a vertex update, we then compare it with the data stored and reduce them if their vertex IDs are the same. Otherwise, the vertex update will be sent to the next stage until an update with the same vertex ID or an empty register is found. Meanwhile, the register in the first stage will output a vertex update (to be routed) to the nearby RUs. The execution of the registers in the same column is fully pipelined for preserving high efficiency.

Example. Figure 11(b) illustrates a workflow of an aggregation pipeline using a 2×2 register array. Initially, suppose there are two vertex updates V_1 and V_3 stored in the first column of the register array and two vertex updates V_2 and V_4 stored in the second column. When a new vertex update V_3' is received, it will be sent to the first column based on the hash value of its vertex ID. In the first cycle, the register in the first stage compares V_3' with the data already stored (i.e., V_1). Since their vertex IDs are different, V_3' is pipelined into the second stage. In the second cycle, since two vertex IDs are the same (i.e., 3), the register in the second stage reduces V_3' with the data stored, and then stores the results V_3''. When reading a vertex update, the registers in the same column

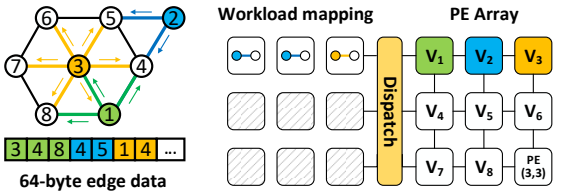


Figure 12. An example of the degree-aware scheduling mechanism

proceed in a systolic manner. Specifically, each register in the i-th stage sends its data to the register in the $(i-1)$-th stage and receives the data sent from the $(i+1)$-th stage. As shown in Figure 11(b), the register in the second stage sends V_4 to the register in the first stage, which further uses V_4 to replace the local V_2 and sends V_2 to the output.

C. Improving Load Imbalance in Scatter Phase

Preserving efficiency is not just about communication efficiency but also load balance. Most vertices have only a few edges in real-world graphs, resulting in a significant gap between the low workload of low-degree vertices and the high parallelism of PEs in the Scatter phase. When processing a low-degree vertex, there may be only several workloads, causing most PEs in a row to be idle. In addition, the number of active vertices can vary drastically in different iterations when processing real-world graphs. As each PE processes only active vertices stored in their local SPDs in the Apply phase, the workloads across different PEs can therefore vary significantly. We then present two solutions to improve load imbalance for Scatter (described below) and Apply (discussed latter in Section IV-D).

Degree-Aware Scheduling. For Scatter phase, we observe that a large set of active vertices usually has continuous memory addresses for its associated edges. For example, all vertices in PageRank are active in each iteration, and hence, all edges are processed sequentially. Therefore, when we access the edges of a low-degree vertex, the edges of the next active vertex very likely reside in the same fetched 64-byte cacheline data. Based on this observation, we propose a degree-aware scheduling mechanism to process multiple low-degree active vertices and their edges simultaneously to prevent the starvation of PEs.

When a VDU receives active vertices from a row of PEs, it accesses their edge addresses and properties from VPref and EPref. Once the 64-byte edge data is received, EDU compares the edge addresses of active vertices with the addresses of the edges accessed to check their connectivity. All active vertices connected to the edges accessed will be scheduled and dispatched along with associated edges to one row of PEs. Consider the example in Figure 12, where two vertices V_2 and V_3 will be scheduled since their edges are stored in the same 64-byte data. This prevents the PE in yellow from starvation. For high-degree vertices, we schedule only one vertex and some of its edges at a time based on the row size of a PE array (i.e., 16 in ScalaGraph) in each cycle. By simultaneously dispatching the workloads

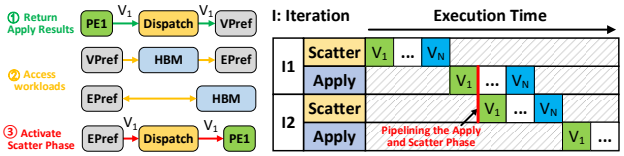


Figure 13. An example illustrating the inter-phase pipelining mechanism

of multiple low-degree vertices, ScalaGraph achieves high PE utilization for real-world graphs.

Hardware Implementation. Dispatching edge workloads of multiple vertices may involve a full connection between 16 inputs (i.e., 64-byte data) and 16 outputs (i.e., a row of PEs), complicating the hardware implementation. We simplify the hardware design by pre-processing edge data. The basic idea is to change the edge layout order of each vertex in the CSR array to make sure that the relative location of an edge in a cacheline is equal to the column index of the PE dispatched. Given K PEs in a row, we maintain K FIFOs for each vertex and store the edges based on the hash values of destination vertices. The edges are shuffled to form a new edge-list by accessing these FIFOs in a round-robin manner. The complexity for this algorithm is O($|E|$), which is the same as that for the format transformation from the edge list to the CSR format.

D. Improving Load Imbalance in Apply Phase

Inter-Phase Pipelining. For Apply phase, we propose a pipelined execution model between Apply and Scatter phases. The key observation is that the updated results in Apply phase in an iteration are the inputs (i.e., active vertices) of Scatter phase in the next iteration. Originally, Scatter phase starts only when Apply phase in the last iteration finishes writing back all active vertices. In our pipelined model, we send the new vertex property to the dispatcher module when a vertex finishes in Apply phase. The Scatter phase can then start without having to wait for the entire active vertex list to finish execution, preventing GUs from starvation.

Workflow Example. Figure 13 gives an illustrative example. After V_1 finishes the Apply phase in the first iteration, the dispatcher stores its computation results, and accesses its edge addresses and edge workloads via the vertex prefetcher and edge prefetcher, respectively. After receiving the edge workloads from HBM, the dispatcher immediately starts the Scatter phase of V_1 in the second iteration without waiting for the completion of the entire Apply phase in the first iteration. As V_1 may be updated by V_N, the Apply phase will start only after the entire Scatter phase has finished. The inter-phase pipelining mechanism improves load imbalance with a fine-grained pipelining scheme.

Limitation. A potential limitation can be data conflicts where a vertex with a smaller ID in the Scatter phase may update the vertices with larger IDs in the Apply phase of the previous iterations. Fortunately, we observe that many graph algorithms (e.g., BFS, SSSP, and CC)

Table III
GRAPH DATASETS USED IN THE EXPERIMENTS

Graph	|Vertex|	|Edge|	Description
Pokec (PK) [41]	1.6M	30.6M	Pokec Social
LiveJournal (LJ) [41]	4.8M	68.9M	Follower
Orkut (OR) [41]	3.0M	234.3M	Orkut Social
RMAT24 (RM) [42]	16.7M	536.8M	Synthetic Graph
Twitter (TW) [1]	41.6M	1468.4M	Twitter Social

update vertex properties monotonically. In this case, our inter-phase pipelining mechanism will not impact their final results and therefore can be safely used. However, for other non-monotonic algorithms (e.g., PageRank), the inter-phase pipelining mechanism is disabled to preserve correctness.

V. EVALUATION

In this section, we evaluate the efficiency and effectiveness of ScalaGraph against the state-of-the-art.

A. Experimental Setup

ScalaGraph Settings. We implement ScalaGraph on a Xilinx Alveo U280 FPGA accelerator card, consisting of a XCU280 FPGA chip. The FPGA provides 1.3M LUTs, 2.6M Registers, 9M BRAM resources, and two 4GB HBM2 stacks with 460GB/s memory bandwidth in total. ScalaGraph contains two tiles with each connecting to one HBM stack. We use BRAM resources to implement a 6MB scratchpad memory, which is evenly sliced to all PEs. The clock rate of ScalaGraph are obtained using Xilinx Vivado 2019.1. We conservatively use 250MHz in our experiments.

Baselines. We compare ScalaGraph with a state-of-the-art graph accelerator GraphDynS [21] and a GPU-based system Gunrock [31]. AccuGraph is excluded since it is consistently inferior to GraphDyns in both performance and scalability. All experiments are performed on a machine with a 14-core Intel E5-2680v4 processor, 128GB DRAM, an NVIDIA V100 GPU (32GB HBM2 with 900GB/s memory bandwidth), and a Xilinx Alveo U280 FPGA.

Since GraphDyns is not open-sourced, we prototype it on FPGA. As discussed in Section II-B, GraphDyns fails to scale with 256 PEs due to its high hardware complexity. In all possible settings, GraphDyns obtains best performance results in the case of 128 PEs. Therefore, we implement GraphDyns with 128 PEs connected via a 128-radix crossbar running at its highest frequency of 100MHz (referred to as **GraphDyns-128**). For ScalaGraph, we also implement two accelerator versions which are equipped with 128 PEs (referred to as **ScalaGraph-128**) and 512 PEs (referred to as **ScalaGraph-512**), respectively.

To demonstrate the effectiveness of ScalaGraph in the case of a large number of PEs, we also extend GraphDyns by using a distributed on-chip memory model to support 512 PEs for apple-to-apple comparison purposes. However, directly replacing the crossbar in GraphDyns with a mesh NoC can even cause performance slowdown (around 1.98 $\times$) against ScalaGraph-128 due to significantly increased NoC communications. Therefore, we implement GraphDyns with four mesh-connected tiles with each consisting of 128 crossbar-connected PEs (referred to as **GraphDyns-512**).

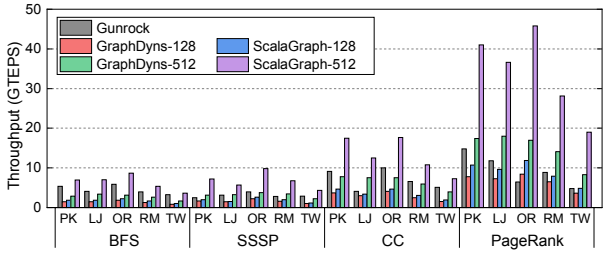


Figure 14. Throughput of ScalaGraph against Gunrock and GraphDyns

Datasets and Algorithms. We consider four representative graph algorithms, i.e., *BFS*, *SSSP*, *CC*, and *PageRank*, performed on both real-world and synthetic graphs in Table III. All graphs are directed and stored in CSR format. For SSSP that runs on weighted graphs, each edge of a graph is associated with a random integer between 0 and 255.

B. Overall Performance

Figure 14 shows the throughput results of ScalaGraph against Gunrock and GraphDyns. The throughput refers to *giga-traversed edge per second* (GTEPS).

ScalaGraph vs. Gunrock. ScalaGraph outperforms Gunrock by an average of 3.2$\times$ throughput. The benefits mainly come from reduced off-chip communications via efficient memory specializations. ScalaGraph reduces 52.2% memory accesses on average due to improved memory efficiency with better locality. Also, ScalaGraph eliminates atomic stalls by avoiding concurrent updates on the same vertex, which can often take more than 15% execution time of GPU-based graph systems. Compared to other graph applications, BFS achieves the smallest speedups due to the power-law degree distribution of real-world graphs. The number of edge workloads in many iterations is too small to fully utilize PEs. In contrast, PageRank offers the highest speedups since all edges are processed in each iteration, showing massive parallelism that can be exploited.

ScalaGraph vs. GraphDyns. In the case of 128 PEs, ScalaGraph achieves only 1.2$\times$ speedup over GraphDyns on average. The main reason lies in the essential difference in the communication efficiency between the centralized and distributed on-chip memory adopted. While the centralized on-chip memory can finish routing in one execution cycle, the distributed one achieves high scalability at the expense of high routing latency. Therefore, when the computation parallelism (i.e., the number of PEs) is relatively low, graph accelerators may not enjoy too much performance benefits from the distributed on-chip memory. However, the benefit achieved by ScalaGraph can quickly increase to 4.6$\times$ when ScalaGraph scales to 512 PEs, demonstrating the potential of distributed on-chip memory for massively parallel graph processing. Even if we extend GraphDyns to 512 PEs via a mesh network similar to ScalaGraph, ScalaGraph-512 is still

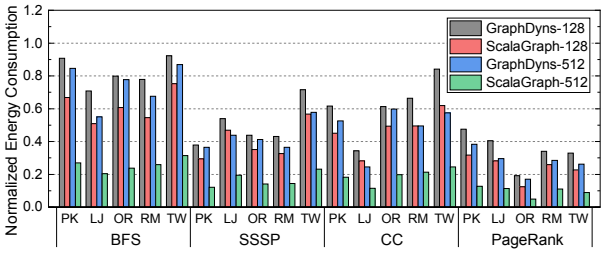


Figure 15. Energy consumption of ScalaGraph against GraphDyns. All results are normalized to Gunrock.

Accelerators	LUT	REG	BRAM
GraphDyns-128	22.8%	11.6%	74.7%
ScalaGraph-128	10.9%	6.4%	70.8%
GraphDyns-512	85.1%	43.8%	76.1%
ScalaGraph-512	39.2%	22.9%	73.2%

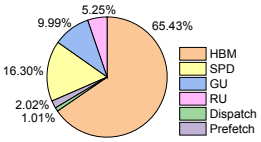
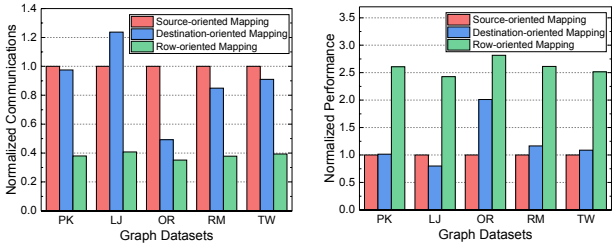


Figure 16. Resource utilization and power breakdown of ScalaGraph

2.2× faster on average than GraphDyns-512. This is because GraphDyns-512 is bottlenecked by frequent on-chip communications between tiles, which are resolved effectively by row-oriented mapping and aggregation pipeline adopted in ScalaGraph (as will be discussed in Section V-C).

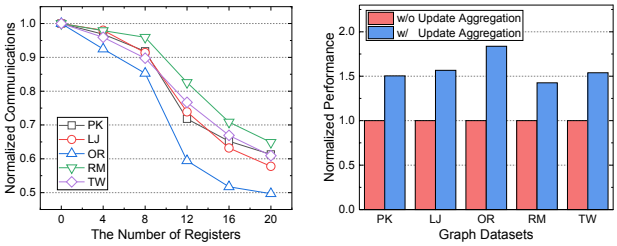
Energy Consumption. Figure 15 shows further the energy consumption results including HBM. The power of the host server is excluded. We use NVIDIA System Management Interface [43] to obtain the energy consumption of GPU. For GraphDyns and ScalaGraph, we use the `query` command provided in Xilinx Board Utility [44] to obtain their energy consumption on FPGA. Compared to Gunrock, ScalaGraph-512 reduces energy by 7.1× on average. The main reason lies in a significant reduction achieved for the amount of off-chip memory communications incurred due to efficient on-chip data management of random vertex accesses in ScalaGraph. Compared to GraphDyns-128, ScalaGraph-128 reduces energy by 1.3× only on average, since the increased communication overhead caused by the Mesh network outweighs the benefits achieved at a small parallelism scale. ScalaGraph-512 can reduce energy by 3.3× and 2.8× against GraphDyns-128 and GraphDyns-512, respectively, due to its high scalability and our energy-efficient NoC mapping and scheduling.

Resource Utilization and Power. It is not easy to compare the areas of two FPGA implementations directly. We have thus opted to use hardware resource utilization instead. Figure 16 shows the resource utilization and power breakdown of ScalaGraph. In the case when the same number of PEs is implemented, ScalaGraph requires 2.1× fewer LUTs and 1.8× fewer REGs than GraphDyns. To evaluate the power of each hardware module, we use Xilinx Vivado-2019.1 under the default toggle rate. Specifically, the off-chip HBM consumes most (∼65.43%) of the energy. The NoC, including RU and communication links, consumes only 5.25% of the total energy. In particular, in the case of 128 PEs, the NoC used in ScalaGraph takes only 53.5% of the power consumed by the crossbar used in GraphDyns.



(a) Normalized Communications (b) Normalized Performance

Figure 17. Effectiveness of the row-oriented mapping in terms of (a) communication and (b) performance. Results are collected by running PageRank where all edges are active (to avoid significant load imbalance).



(a) Normalized Communications (b) Normalized Performance

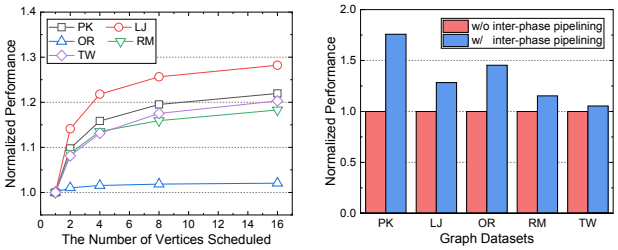
Figure 18. Effectiveness of the update aggregation in terms of (a) the amount of communications incurred under different numbers of registers and (b) performance with and without applying update aggregation

C. Effectiveness of Communication Optimizations

Row-oriented Mapping. Figure 17 shows the effectiveness of our *row-oriented mapping* (**ROM**) against *source-oriented mapping* (**SOM**) and *destination-oriented mapping* (**DOM**). Specifically, DOM needs to store vertex replicas in all PEs, requiring a total space that significantly exceeds the BRAM capacity of the FPGA used. Therefore, the results of DOM are collected from a cycle-accurate accelerator with a large on-chip memory, while the results of ROM and SOM are collected by running ScalaGraph on FPGA.

Compared to SOM, ROM reduces the amount of NoC communications by 61.7% on average since all inter-PE communications cross columns are eliminated. Specifically, the average packet routing latency of SOM is 15.6 cycles, which can be reduced to 5.9 cycles by ROM. Thus, ROM further reduces the execution time by 2.6× on average. The speedups and NoC communication reduction of ROM can be 2× higher than SOM because of two 16 × 16 tiles adopted in ScalaGraph. In this context, ROM can dispatch the edge workloads to both rows of two tiles with an on-chip communication reduction of $\frac{2}{3}$ and a speedup of 3× at most in theory. Compared to DOM, ROM reduces the amount of NoC communications by 28.6% ∼ 67.0%. In general, graphs with higher average vertex degrees obtain fewer benefits. This is because that DOM incurs extra vertex communications in Apply. As the average degree increases, the impact of increased vertex communications can be mitigated to obtain more performance benefits.

Update Aggregation. Figure 18(a) shows the NoC communications of ScalaGraph with different registers used in



(a) Degree-aware Scheduling (b) Inter-phase Pipelining

Figure 19. Performance of ScalaGraph (a) with different numbers of vertices processed simultaneously and (b) with/without inter-phase pipelining

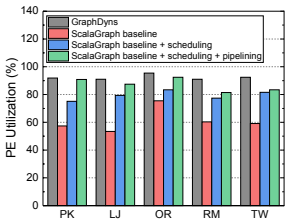


Figure 20. PE utilization between GraphDyns and ScalaGraph

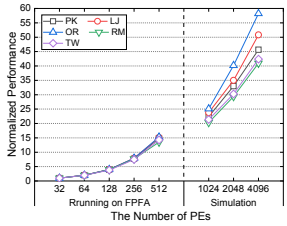


Figure 21. Performance of ScalaGraph with #PEs scaling to 4096

the update aggregation pipeline. The absence of registers means that the aggregation pipeline is equivalent to a FIFO. With more registers, ScalaGraph has a better chance to find the same vertex to be reduced, thus decreasing the required on-chip communications by up to 50.3%. When the number of registers is smaller than 12, update aggregation can even achieve super-linear NoC communication reduction for all graphs since the routing conflicts can be significantly reduced as the number of registers increases. However, when the number of registers increases further from 12 to 20, this effect will be reduced. Consider hardware complexity, we use 16 registers by default. The update aggregation coalesces only the packets destined to the same vertex with the same route path, there improving little routing latency but reducing the NoC communications significantly. As shown in Figure 18(b), update aggregation offers a speedup of 1.57× on average.

D. Effectiveness of Load-balance Optimizations

Degree-Aware Scheduling. Figure 19(a) shows the performance of ScalaGraph by changing the maximal number of vertices that can be simultaneously processed. It can be observed that the performance increases gradually as the number of vertices scheduled increases. When 16 vertices are simultaneously scheduled, the degree-aware scheduling mechanism can achieve 1.02× ∼ 1.28× speedups over a **baseline** that schedules one vertex at a time. Overall, the lower degree a graph has, the more it can benefit from the degree-aware scheduling mechanism. This is because the edge workloads in a low-degree graph are more difficult to utilize massive PEs fully.

Inter-Phase Pipelining. We further characterize the performance of ScalaGraph with and without using our inter-phase pipelining mechanism. Figure 19(b) shows the performance of ScalaGraph on CC. We see that our inter-phase pipelining mechanism can offer 1.05× ∼ 1.76× speedups. In particular, TW shows the smallest benefits since this mechanism is sensitive to partitioning. For a large graph (e.g., TW) whose vertex properties cannot be stored on-chip, it will be sliced into several partitions. When processing an edge whose two associated vertices are stored in different partitions, our mechanism has little effect since the updated vertex properties in an Apply phase cannot be used in the subsequent Scatter phase. Thus, a graph with fewer vertices can benefit more from this mechanism.

Table IV
MAXIMAL FREQUENCY (MHZ) OF SCALGRAPH AGAINST GRAPHDYNS WITH DIFFERENT NUMBER OF PES. '-' DENOTES A SYNTHESIS FAILURE.

PE	32	64	128	256	512	1024
ScalaGraph	304	293	292	285	274	258
GraphDyns	270	227	112	-	-	-

Overall Load Balance. Figure 20 shows the average PE utilization of ScalaGraph against GraphDyns. The higher PE utilization an accelerator has, the better load balance it can achieve. To remove the impact of the mesh network used in GraphDyns-512, we compare only ScalaGraph-128 with GraphDyns-128. We see that ScalaGraph-128 has lower PE utilization than GraphDyns-128, because the central PEs in the Mesh network may be frequently communicated. With our load-balance-related optimizations, the PE utilization of ScalaGraph (87.2%) can be impressive, which is only slightly lower than that of GraphDyns (92.3%). Considering together with a significantly higher frequency, ScalaGraph can therefore yield better performance than GraphDyns.

E. Scalability

We then investigate the scalability of ScalaGraph with different number of PEs ranging from 32 to 1,024. In the case of 32 PEs, each tile in ScalaGraph will contain a 16×1 PE matrix. When increasing the number of PEs, we iteratively add one column to the PE matrix in each tile. When the number of PEs exceeds 1,024, the LUT resources on FPGA will be exhausted. Thus, the configurations with more than 1,024 PEs are excluded. As shown in Table IV, compared to GraphDyns that can scale to a maximum of 128 PEs with only 112MHz, ScalaGraph can support at least 1,024 PEs while preserving a high frequency of 258.5MHz.

Figure 21 further shows the performance of ScalaGraph running at 250MHz frequency with different PE counts. ScalaGraph can achieve nearly linear speedups when the number of PEs is smaller than 512, showing ideal scalability. When the number of PEs increases further from 512 to 1024, which is not presented in Figure 21, ScalaGraph-1024 achieves only 1.16× speedup against ScalaGraph-512 since the available off-chip bandwidth that U280 FPGA can offer is saturated and becomes a performance bottleneck.

In fact, the off-chip bandwidth of today's memory devices can be far more than 460GB/s that U280 FPGA can

offer [25], [45]. To further characterize scalability with over 1,024 PEs, we architect a cycle-accurate simulator running at 250MHz with sufficient off-chip bandwidth, the memory latency of which is collected from running GraphDyns-512 on U280. As is shown, in the case of PEs exceeding 1,024, the performance of ScalaGraph can be improved by 1.47× on average once the number of PEs has a 2× increase, showing good scalability.

VI. Related Work

Graph Processing Systems. Among prior efforts on graph-specific systems [8], [11], [46]–[49], techniques designed for distributed graph processing are most related to our distributed on-chip memory model. PowerGraph [9] proposes a vertex-cut partition to achieve load balance. Gemini [10] adopts a hybrid communication model that enables accessing remote graph data efficiently. These techniques are efficient on general-purpose cores, but need extra storage or complicated scheduling, making it difficult to improve the scalability of graph accelerators that are concerned with not only scalability but also hardware complexity.

GPU-based Graph Processing. Upon thousands of cores, GPU-based graph systems [13], [31], [50]–[53] are qualified in making full use of memory-level parallelism and achieve orders of magnitude performance improvement over CPU-based graph systems. For example, Gunrock represents a novel data-centric abstraction that allows programmers to develop graph primitives at a high-level abstraction while retaining efficient utilization of GPU resources. However, its performance still suffers due to irregular memory accesses in graph processing, while ScalaGraph can offer considerable benefits further with reduced off-chip communications, achieved by sophisticated on-chip memory designs.

Graph Processing Accelerators. There exist a wide spectrum of graph accelerators [18], [19], [22] for improving off-chip memory efficiency for graph processing. HATS [30] proposes a graph-aware prefetcher to reduce random off-chip memory accesses. PolyGraph [54] uses dataflow flexibility to support different graph processing optimizations on spatial architectures. GraphDyns [21] represents a new programming model to extract data dependencies in graph processing dynamically. Based on the run-time dependency information, GraphDyns adopts a load-balanced scheduling mechanism, a specialized prefetcher to precisely prefetch off-chip edge data, and a vectorization technique to improve on-chip vertex access. AccuGraph [20] represents a parallel accumulator for processing multiple memory operations of the same vertex in parallel efficiently. To further enhance parallelism, it also includes a specialized mechanism to allow the on-chip memory to process the requests in an out-of-order manner. In contrast, ScalaGraph focuses on improving the scalability of on-chip memory, which is orthogonal to these earlier efforts.

There are also optimizations in reducing the hardware overhead of centralized on-chip memory. GraphPulse [24] uses a multi-stage crossbar to reduce crossbar radix. Chronos [32] multiplexes mutiple PEs into one crossbar port to reduce hardware complexity. Their scalability has improved at a small scale, but still suffers significantly when a large number of PEs is used, as discussed in Section III-A. In contrast, ScalaGraph supports achieving near-optimal scaling with respect to the number of PE while retaining a low hardware complexity.

Spatial Architectures. Many coarse-grained spatial architectures are adopted by NN accelerators [55], [56]. Eyeriss [57] uses a novel dataflow mapping for spatial architectures to achieve energy-efficient data movement. DSAGEN [58] automates a software-hardware co-design process on spatial architectures. These studies are highly scalable for regular applications, but may perform poorly on irregular graph processing with sparse data due to excessive random memory accesses. In contrast, ScalaGraph is designed for graph-specific applications with improved on-chip memory scalablity.

VII. Conclusion

We introduce ScalaGraph, a scalable graph accelerator, which enables massively parallel graph processing. ScalaGraph is novel with a distributed on-chip memory hierarchy that can be scaled easily to support more than thousands of PEs effectively and efficiently. ScalaGraph also adopts a row-oriented mapping and a pipelined data aggregation for minimizing NoC communications. Also, software-hardware co-designs are proposed to improve load imbalance. Our evaluation shows that ScalaGraph on a modest configuration outperforms a state-of-the-art GPU-based system Gunrock and a graph accelerator GraphDyns by 3.2× and 2.2×, with nearly linear scalability with respect to the number of PEs.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments. This work is supported by the National Natural Science Foundation of China under Grant No. 61832006, 61825202, and 62072195. The correspondence of this paper should be addressed to Long Zheng.

References

[1] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of International Conference on World Wide Web (WWW)*, 2010, pp. 591–600.

[2] G. Linden, B. Smith, and J. York, "Amazon. com recommendations: Item-to-item collaborative filtering," *IEEE Internet Computing*, vol. 7, no. 1, pp. 76–80, 2003.

[3] T. Aittokallio and B. Schwikowski, "Graph-based methods for analysing networks in cell biology," *Briefings in Bioinformatics*, vol. 7, no. 3, pp. 243–255, 2006.

[4] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 1–10.

[5] H. Liu and H. H. Huang, "Enterprise: Breadth-first graph traversal on GPUs," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015, pp. 1–12.
[6] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *Proceedings of Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018, pp. 393–404.
[7] J. Shun, "Practical parallel hypergraph algorithms," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2020, pp. 232–249.
[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010, pp. 135–146.
[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 17–30.
[10] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 301–316.
[11] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of European Conference on Computer Systems (EuroSys)*, 2017, pp. 527–543.
[12] K. Vora, "Lumos: Dependency-driven disk-based graph processing," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2019, pp. 429–442.
[13] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: minimizing data transfer during out-of-GPU-memory graph processing," in *Proceedings of European Conference on Computer Systems (EuroSys)*, 2020, pp. 12:1–12:16.
[14] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, "How well do graph-processing platforms perform? an empirical performance evaluation and analysis," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2014, pp. 395–404.
[15] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, 2015, pp. 56–65.
[16] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: understanding graph computing in the context of industrial solutions," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015, pp. 1–12.
[17] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 373–386.
[18] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.
[19] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-FPGA architecture," in *Proceedings of International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 217–226.
[20] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He, "An efficient graph accelerator with parallel data conflict management," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2018, pp. 8:1–8:12.
[21] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach," in *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 615–628.
[22] Y. Yang, Z. Li, Y. Deng, Z. Liu, S. Yin, S. Wei, and L. Liu, "Graphabcd: scaling out graph analytics with asynchronous block coordinate descent," in *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 419–432.
[23] P. Yao, L. Zheng, Z. Zeng, Y. Huang, C. Gui, X. Liao, H. Jin, and J. Xue, "A locality-aware energy-efficient accelerator for graph mining applications," in *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 895–907.
[24] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 908–921.
[25] NVIDIA, "Nvdia A100 tensor core GPU," https://www.nvidia.com/en-us/data-center/a100/, 2021.
[26] GraphCore, "Graphcore IPU processors," https://www.graphcore.ai/products/ipu, 2021.
[27] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "Thundergp: HLS-based graph processing framework on FPGAs," in *Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2021, pp. 69–80.
[28] S. Beamer, K. Asanović, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 820–831.
[29] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 166–177.
[30] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 1–14.
[31] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016, pp. 1–12.
[32] M. Abeydeera and D. Sanchez, "Chronos: Efficient speculative parallelism for accelerators," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1247–1262.
[33] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *Proceedings of Interna-*

tional Symposium on High Performance Computer Architecture (HPCA), 2018, pp. 724–736.

[34] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: leveraging gustavson's algorithm to accelerate sparse matrix multiplication," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 687–701.

[35] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 7, no. 2, pp. 279–301, 1989.

[36] D. Nassimi and S. Sahni, "A self-routing benes network and parallel permutation algorithms," *IEEE Transactions on Computers*, vol. 30, no. 05, pp. 332–340, 1981.

[37] G. C. Chow, P. Grigoras, P. Burovskiy, and W. Luk, "An efficient sparse conjugate gradient solver using a beneš permutation network," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–7.

[38] Y. Jiang, "Design and implementation of benes/clos on-chip interconnection networks," Ph.D. dissertation, University of Nevada, Las Vegas, USA, 2016.

[39] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for PIM-based graph processing with efficient data partition," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 544–557.

[40] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable PIM-based graph processing," in *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 712–725.

[41] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[42] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[43] Nvidia, "Nvidia system management interface," https://developer.nvidia.com/nvidia-system-management-interface, 2021.

[44] Xilinx, "Xilinx board utility," https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/xilinx-board-swiss-army-knife-utility-ufa1504034339078.html, 2019.

[45] S. hynix, "Hbm2e," https://product.skhynix.com/products/dram/hbm/hbm2e.go, 2021.

[46] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 31–46.

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

[48] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013, pp. 135–146.

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

[50] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 6, pp. 1543–1552, 2013.

[51] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: Vertex-centric graph processing on GPUs," in *Proceedings of the International Symposium on High-performance Parallel And Distributed Computing (HPDC)*, 2014, pp. 239–252.

[52] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2017, pp. 195–207.

[53] L. Zheng, X. Li, Y. Zheng, Y. Huang, X. Liao, H. Jin, J. Xue, Z. Shao, and Q.-S. Hua, "Scaph: Scalable GPU-accelerated graph processing with value-driven differential scheduling," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2020, pp. 573–588.

[54] V. Dadu, S. Liu, and T. Nowatzki, "Polygraph: Exposing the value of flexibility for graph processing accelerators," in *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[55] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 92–104.

[56] H. Yoo, S. Park, K. Bong, D. Shin, J. Lee, and S. Choi, "A 1.93 tops/w scalable deep learning/inference processor with tetra-parallel mimd architecture for big data applications," in *Proceedings of IEEE International Solid-state Circuits Conference (ISSCC)*, 2015, pp. 80–81.

[57] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2016, p. 367–379.

[58] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 268–281.