

Multi-node Acceleration for Large-scale GCNs

Gongjian Sun, Mingyu Yan, Duo Wang, Han Li, Wenming Li,
Xiaochun Ye, Dongrui Fan, *Senior Member, IEEE* and Yuan Xie, *Fellow, IEEE*

Abstract—Limited by the memory capacity and compute power, single-node graph convolutional neural network (GCN) accelerators cannot complete the execution of GCNs within a reasonable amount of time, due to the explosive size of graphs nowadays. Thus, large-scale GCNs call for a multi-node acceleration system (MultiAccSys) like TPU-Pod for large-scale neural networks. In this work, we aim to scale up single-node GCN accelerators to accelerate GCNs on large-scale graphs. We first identify the communication pattern and challenges of multi-node acceleration for GCNs on large-scale graphs. We observe that (1) coarse-grained communication patterns exist in the execution of GCNs in MultiAccSys, which introduces massive amount of redundant network transmissions and off-chip memory accesses; (2) overall, the acceleration of GCNs in MultiAccSys is bandwidth-bound and latency-tolerant. Guided by these two observations, we then propose MultiGCN, the first MultiAccSys for large-scale GCNs that trades network latency for network bandwidth. Specifically, by leveraging the network latency tolerance, we *first* propose a topology-aware multicast mechanism with a one put per multicast message-passing model to reduce transmissions and alleviate network bandwidth requirements. *Second*, we introduce a scatter-based round execution mechanism which cooperates with the multicast mechanism and reduces redundant off-chip memory accesses. Compared to the baseline MultiAccSys, MultiGCN achieves 4~12 \times speedup using only 28%~68% energy, while reducing 32% transmissions and 73% off-chip memory accesses on average. It not only achieves 2.5~8 \times speedup over the state-of-the-art multi-GPU solution, but also scales to large-scale graphs as opposed to single-node GCN accelerators.

Index Terms—Graph neural network, hardware accelerator, multi-node system, processor cluster, communication optimization.

1 INTRODUCTION

GRAPH Convolutional Neural Networks (GCNs) have emerged as a premier paradigm to address the graph learning problem via generalizing the information encoding to graph topologies that can represent extremely complicated relationships [1], [2], [3], [4], [5]. In reality, GCNs have been widely applied in many critical fields such as knowledge inference [6], recommendation system [7], visual reasoning [8], traffic prediction [9], Electronic design automation (EDA) [10], and GCN workloads can be found at many data centers [1], [9].

GCNs typically exhibit a hybrid execution pattern introduced by two distinct execution phases, which hinder the acceleration of GCNs in GPUs [11], [12]. The *Aggregation* phase traverses all vertices and aggregates the feature vectors of neighboring vertices into the current vertex, displaying an irregular execution pattern like graph processing (GP). The *Combination* phase further transforms the feature vector of each vertex into a new one using a multi-layer perceptron (MLP), exhibiting a regular execution pattern like neural network (NN). Such interleaving execution patterns hinder the acceleration of GCNs in GPUs which are inherently optimized for compute-intensive workloads with regular execution pattern [13].

To tackle this hybrid execution pattern, previous efforts [11], [14], [15], [16], [17] propose a series of single-node GCN accelerators. Although these accelerators have achieved great improvement on both performance and energy efficiency compared with

GPUs, they suffer from two following inefficiencies in the processing of large-scale GCNs. *First*, with limited hardware resources from computation and buffering, a single-node accelerator cannot complete the execution of large-scale GCNs within a reasonable amount of time, not to mention that the sizes of real-world graphs keep growing rapidly [1], [18]. *Second*, massive time and energy have to be taken to move data between memory and hard disks [1], because single-node accelerators do not have enough memory to accommodate the entire large-scale graph. Thus, a multi-node acceleration system (MultiAccSys) is highly desirable for large-scale GCNs.

Previous efforts have proposed a series of MultiAccSyses for NNs and GPs, achieving great improvements on both performance and energy efficiency. However, they fail to tackle the unique execution pattern in the multi-node acceleration of GCNs. MultiAccSyses for NNs, e.g., TPU-Pod [19], tailor their hardware datapaths to the regular execution pattern such as coarse-grained regular communication pattern inter-node, to leverage the regularity for high performance of large-scale NN acceleration. Similarly, MultiAccSyses for GPs, e.g., Tesseract [20], tailor their hardware datapaths to the irregular execution pattern of GPs such as fine-grained irregular communication pattern inter node, to alleviate the irregularity for high performance of large-scale GP acceleration. Unfortunately, the multi-node acceleration for GCNs exhibits distinct execution patterns, i.e., the coarse-grained irregular communication inter node and hybrid execution pattern intra node. Such execution patterns make MultiAccSyses ill-suited for GCNs, since they are only designed to either exploit regular execution pattern or alleviate irregular execution pattern.

In this paper, we aim to scale the single-node accelerator to accelerate GCNs on large-scale graphs like TPU-Pod [19], since the performance and energy efficiency of single-node GCN accelerators are significantly higher than that of high-end GPUs.

- G. Sun, M. Yan, D. Wang, H. Li, W. Li, X. Ye, D. Fan are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China (Email: {sungongjian19s, yanmingyu, wangduo18z, lihan-ams, liwenming, yexiaochun, fandr}@ict.ac.cn). G. Sun, M. Yan, D. Wang, H. Li, W. Li, D. Fan are also with University of Chinese Academy of Sciences, Beijing, China. Y. Xie is with University of California, Santa Barbara, California, USA (Email: yuanxie@ece.ucsb.edu). Corresponding author is Mingyu Yan (yanmingyu@ict.ac.cn).

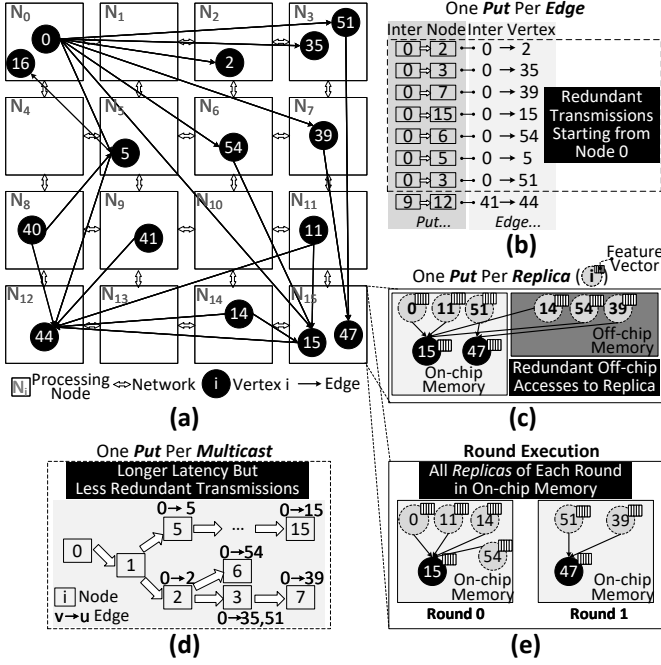


Fig. 1. Comparison between MultiGCN and previous efforts on MultiAccSys for graph processing: (a) Examples of graph and MultiAccSys for GCNs; (b) Disadvantage of MultiAccSys with one put per edge message-passing model; (c) Disadvantage of MultiAccSys with one put per replica message-passing model; (d) and (e) Advantage of MultiGCN based on one put per multicast message-passing model and scatter-based round execution.

To identify the unique execution pattern and the challenges of the multi-node acceleration for GCNs, we characterize the execution of large-scale GCNs in a straightforward design of MultiAccSys for GCNs as shown in Figure 1(a). To ensure an effective characterization, we borrow a well-designed single-node GCN accelerator [11] to design the processing node and a representative message-passing model to alleviate the irregular communication pattern inter node, i.e., one put per edge (OPPE) [20]. We observe irregular coarse-grained communication patterns in the execution of GCNs in MultiAccSys, which introduce massive amount of redundant network transmissions and off-chip memory accesses. This communication pattern is the direct result of the following facts: (1) each transmission between nodes contains a long feature vector of neighbor vertex and (2) it is unpredictable when and to where it needs to be sent due to the irregular connection pattern of neighbors. As a result, the OPPE message-passing model invokes many redundant coarse-grained transmissions because the long feature vector of each vertex must be repeatedly sent to all of its neighboring vertices. However, many of them may be sent to or pass through the same processing node. For example, in Figure 1(b), the feature vector of vertex V_0 in the processing node N_0 is sent to neighboring vertices V_2 , V_{35} , V_{51} , V_{39} ... in processing nodes N_2 , N_3 , N_3 , N_7 ..., respectively. To reduce these redundant transmissions, a one put per replica (OPPR) message-passing model is proposed [21], which only puts one replica of the feature vector to each processing node and shares it with all neighboring vertices in that processing node. However, it is difficult to store thousands of replicas on-chip, which inevitably leads to massive amount of off-chip memory accesses. For example, Figure 1(c) shows that vertices V_{15} and V_{47} in the processing node N_{15} require accesses to the replicas

of V_{14} , V_{54} , and V_{39} , which were stored off-chip after receiving because on-chip memory had been exhausted.

In light of the above challenges, we propose MultiGCN, the first MultiAccSys for GCNs, to accelerate the inference phase of large-scale GCNs by trading network latency for network bandwidth. *First, a topology-aware multicast mechanism with a one put per multicast message-passing model is proposed to alleviate network bandwidth requirements.* As shown in Figure 1(d), by leveraging the latency tolerance we identified as well as the known network and graph topologies, V_0 's feature vector is multicast in the transmission to reduce redundant transmissions. *Second, a scatter-based round execution mechanism is proposed to cooperate with the multicast mechanism, which inherently matches the behaviour of multicast.* Specifically, each processing node scatters the replicas of vertices' feature vectors to remote processing nodes which perform aggregation for their neighboring vertices. Besides, to reduce redundant off-chip memory accesses, the graph is partitioned into a certain number of sub-graphs, one for each execution round, as shown in Figure 1(e). Thus, all replicas of round 0 (i.e., V_0 , V_{11} , V_{14} , and V_{54}) and round 1 (i.e., V_{51} and V_{39}) from remote processing nodes can be stored on-chip until the corresponding round completes. Intra- and inter-round overlap are utilized for higher performance.

The key contributions of this paper are as follows:

- We identify the communication pattern and challenges of multi-node acceleration for large-scale GCNs and observe that: (1) There are irregular coarse-grained communication patterns that invoke massive redundant transmissions and off-chip memory accesses; (2) Execution of GCNs in MultiAccSys is mainly bandwidth-bound and latency-tolerant.
- Accordingly, we propose the first multi-node acceleration system for GCNs which scales single-node GCN accelerators to accelerate GCNs on large-scale graphs by trading network latency for network bandwidth.
- We propose a topology-aware multicast mechanism to reduce redundant transmissions and alleviate network bandwidth requirements, along with a scatter-based round execution mechanism that cooperates with the multicast mechanism to reduce off-chip memory accesses.
- We implement MultiGCN in both RTL and cycle-accurate simulator to demonstrate its advantages. Compared with OPPE-based MultiAccSys, MultiGCN achieves 4~12 \times speedup, and reduces 32% network transmissions as well as 73% off-chip memory accesses on average. Besides, MultiGCN achieves 2.5~8 \times speedup over the multi-GPU solution and can scale to large-scale graph as opposed to single-node GCN accelerators.
- We explore the design of MultiGCN in detail and provide insights into the relationships between hardware parameters, graph characteristics, and architecture techniques.

TABLE 1
Notations used in this paper.

Notation	Explanation
$G = (V, E)$	directed graph G
$V(V)$	(size of) vertex set of graph G
$E(E)$	(size of) edge set of graph G
(i, j) or $e_{i,j}$	edge from vertex i to vertex j
d_v	in-degree of vertex v
N_v	incoming neighbor set of vertex v
$h_v^k(h_v^k)$	(length of) feature vector of vertex v at k -th layer
a_v^k	aggregated result of vertex v at k -th layer

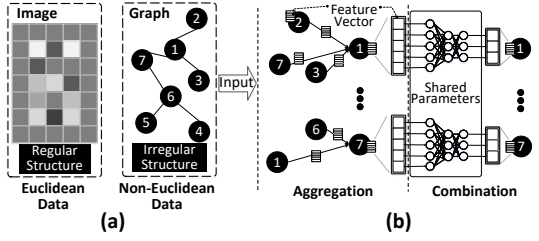


Fig. 2. Illustration examples of (a) graph and (b) GCNs execution.

2 BACKGROUND

GCNs. As shown in Figure 2, typical GCNs take non-euclidean data as input, i.e., graphs which have irregular structure and differ from the grid-structured images as shown in Figure 2(a). GCNs typically consist of several graph convolutional layers, each with two main phases: *Aggregation* and *Combination*, which can be formulated as:

$$a_v^k = \text{Aggregate} \left(h_u^{k-1} \right)_{u \in \{v\} \cup N_v}, h_v^k = \text{Combine} \left(a_v^k \right),$$

where $v \in V$. For clarity, by “node” we refer to the processing nodes in MultiAccSys, and by “vertex” we refer to the elements of the graph vertex set.

In the k -th layer, for each vertex v , the feature vectors h_u^{k-1} of neighboring vertices in N_v are aggregated into a_v^k , as shown in Figure 2(b). Since *Aggregation* phase heavily relies on the graph structure that can be arbitrary and sparse, it suffers from irregular data communication patterns. The *combine* function further transforms a_v^k to generate h_v^k using an MLP in the *Combination* phase. All vertices perform *combine* function using a shared MLP, which results in intensive computation and high data locality in the *Combination* phase. The feature vector h_v^k can be very long, and often up to thousands in the initial feature vectors h_v^0 of all vertices.

Network and Message-passing Model. Multi-node acceleration becomes an inevitable trend due to the ever-increasing demand on computation and storage capability in emerging domains such as deep learning [22]. High-speed network interfaces and well-designed network topologies are the basics of effective multi-node acceleration. Network interfaces include NVLink for GPU interconnection, PCIe for CPU and GPU interconnection, and so on. Network latency is a metric used to evaluate how much time is consumed by the network interface to send a minimal packet from the source to the destination. Network topology includes star, bus, mesh, torus, and so on. For example, a NVIDIA HGX houses 16 H100 GPUs networked together using NVLink and NVSwitch in star topology¹, with a peak communication bandwidth of up to 900GB/s. The message-passing model determines the transfer mode of MultiAccSys, such as the OPPE and OPRR models used in the MultiAccSys of GPs. For each vertex, the OPPE model sends one replica of the feature vector to each neighboring vertex, while the OPRR model only sends one replica of the feature vector to each processing node and shares it with all neighboring vertices in that processing node.

3 MOTIVATION

Inefficiencies of Single-node GCN Accelerators. Previous efforts propose several single-node accelerators for GCN acceleration, achieving significant improvements in both performance

and efficiency compared with GPUs. However, the ever-growing scale of graphs hinders efficient execution of GCNs on single-node accelerators. For example, to tackle the hybrid execution pattern, HyGCN [11] proposes a hybrid architecture for GCNs, which includes two engines respectively tailored to *Aggregation* and *Combination* phases. HyGCN achieves an average $6.5\times$ speedup with $10\times$ energy reduction over high-end GPUs. However, with limited off-chip memory bandwidth, on-chip buffer, and compute resources, a single-node accelerator cannot process large-scale GCNs within a reasonable amount of time, while the scale of real-world graphs continues to grow rapidly [1], [18]. In addition, large-scale graphs demand massive memory, which is hard to satisfy within a single-node accelerator [1], resulting in costly data transfers between the memory and the hard disk. Thus, a MultiAccSys for GCNs is highly desirable.

Inefficiencies of NN and GP MultiAccSyses. Previous efforts propose a series of MultiAccSyses for large-scale NNs and GPs. However, they fail to tackle the unique execution pattern of the multi-node acceleration for GCNs. For example, the designers of TPU-Pod [19] elaborately customize an MultiAccSys for NNs using an NVLink-like inter-node network interface to connect TPU chips. Although TPU-Pod delivers near-linear speedup for large-scale NNs, they are ill-suited for GCNs due to the irregular coarse-grained communication pattern and hybrid execution patterns. Another example is Tesseract [20], whose designers elaborately customize an MultiAccSys for GPs using a well-designed message passing model (i.e., OPPE) to alleviate the fine-grained irregular communication pattern. Although Tesseract achieves significant speedup for large-scale GPs, it invokes massive amounts of redundant coarse-grained transmissions caused by the irregular coarse-grained communication pattern. Although the redundant transmissions can be reduced by introducing the OPRR message-passing model [21], the replicas of feature vectors can be prohibitively large to be stored entirely on-chip, which inevitably leads to many off-chip memory accesses.

Characterization on A Straightforward Design. To identify the communication pattern and challenge of the multi-node acceleration for GCNs, a detailed characterization is conducted on an OPPE-based MultiAccSys and results are shown in Figure 3. Here, we briefly introduce the experimental setup. Please see Section 5 for our detailed evaluation methodology. We use a OPPE-based MultiAccSys consisting of 16 processing nodes, connecting in a manner of 4×4 -sized 2D torus bidirectional network. Each processing node is a variant of the single-node GCN accelerator of previous work [11], having exactly the same hardware resources and one disjoint part of the graph data. The message-passing model used in this MultiAccSys is inspired by the OPPE model, which aims to tackle the irregular communication pattern caused by the irregular graph topology. An in-house cycle-accurate simulator is designed and implemented to measure execution time. We additionally implement the core modules in Verilog and synthesized using Synopsys compilers to check clock frequency requirement and estimate performance metrics.

We observe that the irregular coarse-grained communication patterns exist in the execution of GCNs in MultiAccSys, which introduce massive amounts of redundant network transmissions and off-chip memory accesses. The irregular coarse-grained communication pattern is caused by two reasons: (1) each transmission between node contains a long feature vector of neighbor vertex, with up to hundreds of elements, determined by the size of the input dataset or the number of the MLP’s output neurons, and (2)

1. <https://www.nvidia.cn/data-center/hgx/>

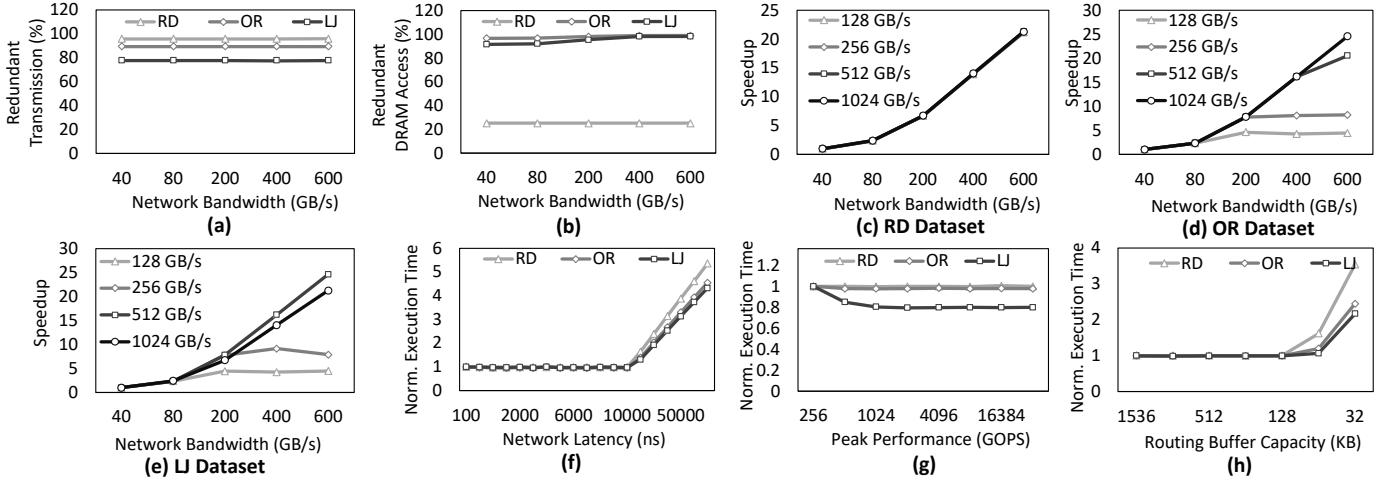


Fig. 3. Results of characterization on OPPE-Based MultiAccSys with 16 processing nodes: (a) Ratio of redundant transmissions to total transmissions across different network bandwidths; (b) Ratio of redundant DRAM accesses to total DRAM accesses across different network bandwidths; Speedup of GCN across different network bandwidths (X-axis) and DRAM bandwidths (4 Lines) on (c) RD, (d) OR, and (e) LJ datasets; Normalized execution time across (f) different network latencies, (g) different peak performances, and (h) different routing buffer capacities.

it is unpredictable when and to where a long feature vector needs to be sent due to the irregular connection pattern of neighbors in graph. As depicted in Figure 3(a) and (b), we observe a vast range of redundant transmissions and DRAM accesses, ranging from 78% to 96% and 25% to 99.9%, respectively. This is because the long feature vectors of each vertex must be repeatedly sent to all of its neighboring vertices, many of which may be sent to or through the same processing nodes. In addition, received feature vectors need to be saved to DRAM upon receipt and loaded from DRAM while in use due to the large number of long feature vectors and limited on-chip cache capacity. These redundancies waste network bandwidth and DRAM bandwidth, which significantly hinders the performance and efficiency on MultiAccSys for GCNs.

We also observe that the acceleration of GCNs in MultiAccSys is bandwidth-bound and latency-tolerant. Figure 3(c), (d), and (e) show that the speedup across different datasets grows almost linearly as network bandwidth increases when DRAM bandwidth is sufficient (i.e., greater than 256 GB/s). This is because neighboring feature vectors with hundreds of elements for each vertex need to be sent and aggregated in a target processing node, following the irregular neighbor connections in *Aggregation* phase, which consumes much network bandwidth for better performance. However, Figure 3(f) shows that the normalized execution time is nearly constant under different network latencies until around 20,000 ns. This value mainly relies on the processing time which is positively correlated with the length of the feature vector and negatively correlated with the DRAM bandwidth of the processing node. This is because the received feature vectors is frequently stored to or load from DRAM as aforementioned. Figure 3(g) shows that the normalized execution time is nearly constant under different peak performances with more than 1024 giga operations per second (GOPS). This is because the low utilization of network bandwidth and DRAM bandwidth become the performance bottleneck. Figure 3(h) shows that the normalized execution time is also nearly constant under different routing buffer capacities above 64 KB. Since the routing buffer is used to buffer the routing packets before they are sent, its capacity relies on the utilized network bandwidth and network latency.

4 MULTIGCN ARCHITECTURE

Guided by the above observations, we propose MultiGCN, an efficient MultiAccSys for large-scale GCNs that trades network latency for network bandwidth.

4.1 Architecture Overview

Figure 4 provides an illustration of the proposed architecture. Noting that MultiGCN does not rely on a particular network topology, we choose a 2D torus topology consisting of 16 processing nodes as our baseline, which is shown in Figure 4(a). For network links we use the NVLINK protocol, which is one of the most widely used high-speed interconnection protocols and usually used between NVIDIA GPUs. A processing node, shown in Figure 4(b), is composed of a compute unit, a router, a receive unit, a send unit, an edge buffer, a scheduler, a loader, an aggregation buffer, a weight buffer, a combination buffer, and DRAM.

The compute unit consists of eight reusable 1×128 systolic arrays. Each processing element (PE) has four registers with two for input, one for output, and one for internal use respectively, and an ALU capable of multiplication and reduction (like MIN, MAX, ADD). The eight systolic arrays work separately, either in combination mode, like a traditional systolic array, or in aggregation mode. In aggregation mode, all PEs follow an identical three-stage pipeline: read two operands from the input registers, perform reduction, and write the result to the output register. Moreover, a real-time scheduling of compute resources between aggregation and combination is implemented in MultiGCN, since all eight reusable systolic arrays can process workloads of both types. Note that although a unified compute unit is used in this work, designs in other single-node accelerators can also be integrated for better efficiency or compatibility.

The router, receive unit, and send unit are used to transfer vertices' feature vectors and neighboring lists. The edge buffer and scheduler are used to efficiently organize computation. Each entry in the edge buffer contains the address of a vertex's feature vector in aggregation buffer and its neighbor list. The feature vector is read via address and aggregated into the intermediate result of vertices in the neighbor list. The aggregation process is recorded aside the intermediate result in the aggregation buffer.

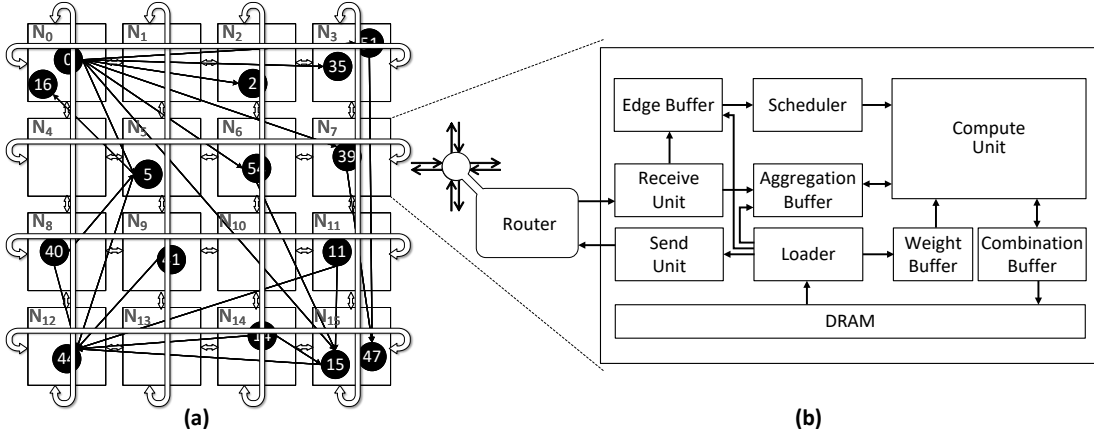


Fig. 4. Architecture of MultiGCN: (a) 2D torus network with 16 processing nodes; (b) Design of processing node.

The weight buffer and combination buffer save the weight matrix and intermediate result for the combination process. The loader loads the meta-data of execution, ID and degree of vertices, feature vectors and edge lists, which the send unit and scheduler ingest to complete execution.

Topology-aware Multicast (Section 4.2). To reduce the requirement of network bandwidth, a topology-aware multicast mechanism with a one put per multicast message-passing model is proposed. Multicast is based on the simple idea that for a vertex v , many processing nodes where v 's neighbors lie in can be satisfied by receiving the same packet containing a replica of v 's feature vector. To efficiently tailor multicast to the communication pattern of GCNs, we design our multicast to be network topology-aware and graph topology-aware. The network topology awareness helps route and split packets guided by routers' status and an explicit list of destination node IDs in the packet. Thus, the packet can be quickly and exactly multicast to all destination nodes. The graph topology awareness is enabled by the offset list and neighbor lists in the packet which are used to share exactly one replica to all neighbors in the same processing node. Although these capabilities introduce extra latency during transmission because of the above additional info in the packets, they help eliminate redundant transmissions and significantly reduce the need for network bandwidth.

Scatter-based Round Execution (Section 4.3). Although the topology-aware multicast mechanism helps reduce the requirements of network bandwidth, its overhead is high for three reasons. *First*, a request-response loop, required by each replica's transmission, significantly increases the design complexity of MultiAccSys for the multicast mechanism. *Second*, the large offset list and neighbor lists are coupled into a single packet, resulting in an unacceptable routing latency in multicast. *Third*, on-chip memory is unable to buffer all thousands of replicas of long feature vectors, which leads to frequent replacements of replicas between on-chip memory and off-chip memory. To this end, a scatter-based round execution mechanism is proposed, which inherently matches the behaviour of multicast. Specifically, each processing node scatters the replicas of vertices' feature vectors to remote processing nodes who perform the `aggregate` function for these vertices' neighboring vertices, so that the request-response loop is eliminated. Besides, the graph is partitioned into multiple sub-graphs, one for each execution round. Thus, the large neighbor lists are partitioned and transmitted over several rounds, avoiding the unacceptable routing latency. Moreover, only a small number

of replicas in each round are stored on-chip until the corresponding round completes, reducing redundant off-chip memory accesses.

4.2 Topology-aware Multicast Mechanism

To enable efficient multicast, we propose a one put per multicast message-passing model based on the DyXY routing algorithm [23] to implement the multicast mechanism for a given network topology and graph topology.

One Put per Multicast. Our one put per multicast model is inspired by multicast on Ethernet. In Ethernet, multicast is a group communication where data transmission is addressed to a group of destination computers simultaneously. In MultiGCN, `put` refers to putting the replica of a vertex's feature vector to its neighboring vertices, while `multicast` means the packet generated by `put` is scattered to all its destinations in Ethernet multicast-like fashion.

The basic routing algorithm we use is DyXY [23], which provides adaptive routing based on congestion status nearby. The algorithm is adaptive in that it makes routing decisions by monitoring the congestion status nearby. It is also deadlock-free and livelock-free since the path of a packet in the network to one of the shortest paths between the source and the destination is limited. If multiple shortest paths are available, the routers will help the packet to choose one of them based on the congestion condition of the network. A stress value, i.e., the occupation ratio of the routing buffer, is used to represent the congestion condition of a router in this work. Each router stores instant stress values for all neighbors, which are updated periodically. The detailed routing algorithm is shown in Algorithm 1 and a routing example for the replica of V_0 's feature vector is depicted in Figure 5(a). See Section 4.3 for details of graph mapping.

To couple the multicast mechanism with the DyXY routing algorithm, step ① in Algorithm 1 is modified to split packets as shown in Algorithm 2. Figure 5(b) shows the packet format used in Algorithm 2, which consists of four parts including the position of the next destination node (x, y), network topology (neighbor-ID (nID) list and its size), graph topology (offset list, neighbor lists), and the replica of feature vector. In the line ② of Algorithm 2, all $nIDs$ in the nID list are transformed into the new coordinate $[x, y]$ by taking the current node (T_x, T_y) as the Origin of coordinates and donated as a set $D = \{[x, y]\}$, when a packet arrives at the destination node (T_x, T_y). Figure 5(c) gives an example for the transformation of nID , new node position $[x, y]$, and fixed node position (x, y) . In this example, N_1 (1, 0) is taken

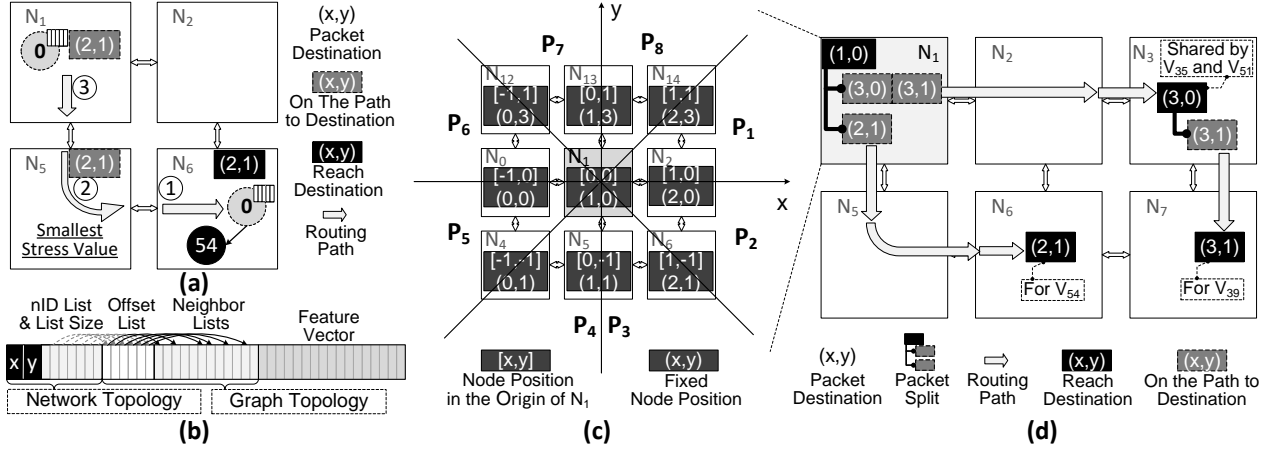


Fig. 5. Illustration of topology-aware multicast based on DyXY routing algorithm: (a) Example of DyXY routing algorithm; (b) Packet format for topology-aware multicast; (c) Packet split of topology-aware multicast in the step 1 of DyXY; (d) One possible multicast of V_0 's feature vector from N_1 (Origin) to V_{35} and V_{51} in N_3 , V_{39} in N_7 , and V_{54} in N_6 .

Algorithm 1: DyXY Routing Algorithm

```

1 foreach packet  $p$  to  $(x,y)$  in routing buffer do
2   ① if  $(x == Tx) \&\& (y == Ty)$  then
3     receive  $p$  in current node  $(Tx, Ty)$ ;
4   end
5   ② else if  $(x == Tx) \parallel (y == Ty)$  then
6     send  $p$  to neighbor on Y-axis or X-axis;
7   end
8   ③ else
9     send  $p$  to neighbor with smallest stress value;
10  end
11 end

```

as the Origin $[0,0]$, and the $[x,y]$ of its neighboring nodes are shown. In line ③ to ⑫, based on the new coordinates, the packet is partitioned into nine parts. Each part has a part of the nID list and neighbor lists, a new offset list, and a complete replica. In line ⑭ to ⑳, these nine parts are received by the current node or sent to the next destination node. Figure 5(d) gives a multicast example based on Algorithm 2 where the replica of V_0 's feature vector is multicast from N_1 to V_{35} and V_{51} in N_3 , V_{39} in N_7 , and V_{54} in N_6 . Specifically, the packet arrives at N_1 (1,0) and is then split into two parts. One is P_1 and consists of one destination including $[1, -1]$ (i.e., N_6 (2, 1)). The other one is P_2 and consists of two destination nodes including $[2, 0]$ (i.e., N_3 (3, 0)) and $[2, -1]$ (i.e., N_7 (3, 1)). Then, the former is sent to N_6 via N_5 . The latter is sent to N_3 via N_2 and is further multicast until the nID list in packet is empty. The packet received by N_3 is further shared by the aggregation of V_{35} and V_{51} according to the neighbor list in the packet. As a result, redundant transmissions are eliminated.

In this process, we have followed the spirit of trading latency for network bandwidth: although the additional info in the packets induce extra transfer latencies, the topology-aware multicast mechanism alleviates the requirement of network bandwidth. However, this also introduces three inefficiencies: high design complexity, low utilization of compute resource, and redundant off-chip memory accesses. *First*, a request-response loop, required by each transmission of the replica, will significantly increase the design complexity. *Second*, the large neighbor lists in the packet cause intensive transmission and unacceptable routing latency, and most of the compute resources become underutilized. *Third*,

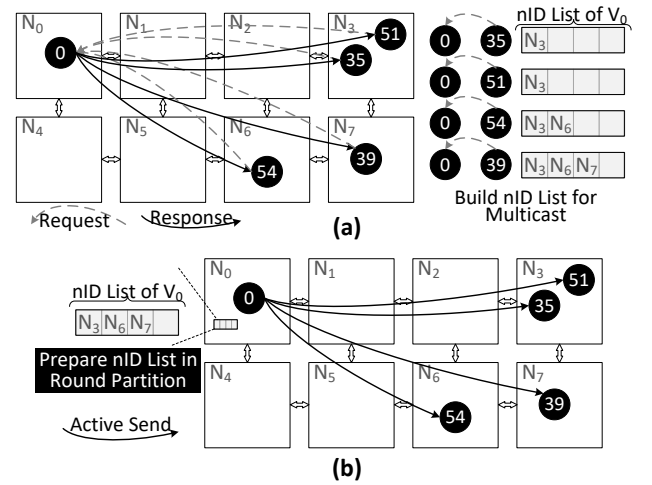


Fig. 6. The execution of `aggregate` function: (a) Gather-based method; (b) Scatter-based method.

limited by the capacity of on-chip memory in each processing node, the large volume of received replicas need to be frequently moved between on-chip memory and off-chip memory. This is because thousands of replicas are received and shared by many vertices' aggregation in each processing node, but it takes a long time to process the long feature vectors, which means most of these replicas need to be first stored in off-chip memory and then reloaded when needed.

4.3 Scatter-based Round Execution Mechanism

To address the above inefficiencies, we propose a scatter-based round execution mechanism that well suits the topology-aware multicast mechanism. The key idea of the scatter-based round execution mechanism is simple but effective: we first partition the graph into a set of sub-graphs and then process one sub-graph per round. In each round, all replicas are kept on-chip until no longer needed. To improve resource utilization, we also implement intra- and inter-round overlaps.

Scatter or Gather. There are two ways to execute the `aggregate` function: gather-based and scatter-based methods. As shown in Figure 6(a), in the gather-based method, each processing node (e.g., N_3, N_6 , and N_7) first requests feature vectors

Algorithm 2: Packet Split of Multicast

```

1 ① if (x == Tx) && (y == Ty) then
2    transform nID list into D = {[x,y]} by taking node (Tx,
    Ty) as the Origin of coordinates;
3    split packet p into the following nine parts:
4    P0 = {[0, 0]} ∩ D ;
5    P1 = {[x, y] | y > 0, y ≤ x} ∩ D ;
6    P2 = {[x, y] | y ≤ 0, y > -x} ∩ D ;
7    P3 = {[x, y] | x > 0, y ≤ -x} ∩ D ;
8    P4 = {[x, y] | x ≤ 0, y < x} ∩ D ;
9    P5 = {[x, y] | y < 0, y ≥ x} ∩ D ;
10   P6 = {[x, y] | y ≥ 0, y > -x} ∩ D ;
11   P7 = {[x, y] | y ≥ -x, x < 0} ∩ D ;
12   P8 = {[x, y] | x ≥ 0, y > x} ∩ D ;
13   receive and share P0 in current node (Tx, Ty);
14   if (P1 ≠ ∅) && (P2 ≠ ∅) then
15     | send P1 ∪ P2 to [MIN(P1.x ∪ P2.x), 0];
16   end
17   else
18     | send P1 to [MIN(P1.x), MIN(P1.y)];
19     | send P2 to [MIN(P2.x), MAX(P2.y)];
20   end
21   if (P3 ≠ ∅) && (P4 ≠ ∅) then
22     | send P3 ∪ P4 to [0, MAX(P3.y ∪ P4.y)];
23   end
24   else
25     | send P3 to [MIN(P3.x), MAX(P3.y)];
26     | send P4 to [MAX(P4.x), MAX(P4.y)];
27   end
28   if (P5 ≠ ∅) && (P6 ≠ ∅) then
29     | send P5 ∪ P6 to [MAX(P5.x ∪ P6.x), 0];
30   end
31   else
32     | send P5 to [MAX(P5.x), MAX(P5.y)];
33     | send P6 to [MAX(P6.x), MIN(P6.y)];
34   end
35   if (P7 ≠ ∅) && (P8 ≠ ∅) then
36     | send P7 ∪ P8 to [0, MIN(P7.y ∪ P8.y)];
37   end
38   else
39     | send P7 to [MAX(P7.x), MIN(P7.y)];
40     | send P8 to [MIN(P8.x), MIN(P8.y)];
41   end
42 end

```

of neighboring vertices (e.g., V_0) for each vertex (e.g., V_{35} , V_{51} , V_{54} , and V_{39}) from the remote processing node (e.g., N_0) and then waits for the responded feature vectors. Afterwards, the replicas of feature vectors are aggregated in the processing node of the requester (e.g., N_3 , N_6 , and N_7). As a result, a request-response loop for each transmission of the replica is introduced. Besides, to support multicast, the remote processing node needs to collect requests for each vertex's feature vector to build an nID list and then performs multicast based on this nID list.

As shown in Figure 6(b), in the scatter-based method, each processing node has an nID list derived from round partition (described in the next paragraph). Each processing node (e.g., N_0) actively sends the feature vector of each vertex (e.g., V_0) to the remote processing nodes (e.g., N_3 , N_6 , and N_7) where the out-going neighboring vertices (e.g., V_{35} , V_{51} , V_{54} , and V_{39}) reside. Then, the feature vectors are aggregated in the remote processing nodes. As a result, message passing only happens in a single direction. From the above analysis, it can be seen that the scatter-based method inherently matches the behavior of topology-aware multicast mechanism, helping eliminate the

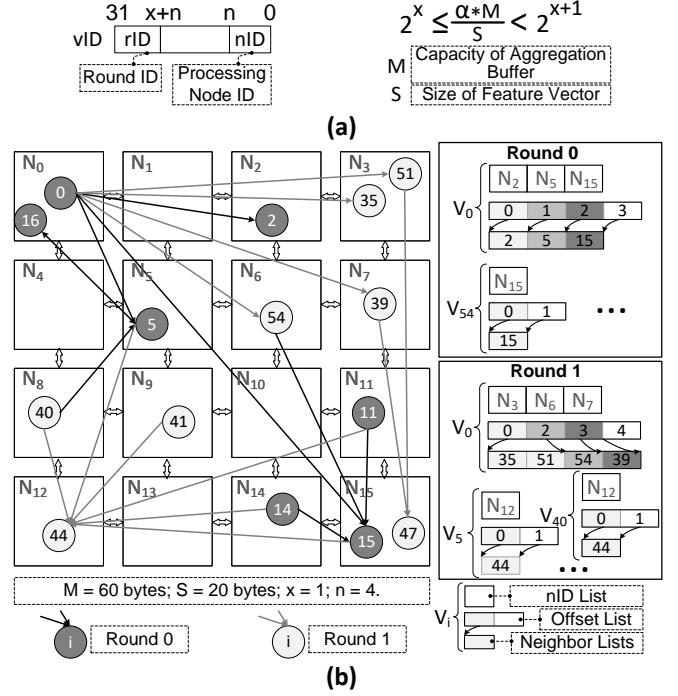


Fig. 7. Illustration of round partition: (a) Generation of round ID; (b) Example of round partition.

request-response loop, and thus achieves lower design complexity than the gather-based method. Hence, we employ the scatter-based method for MultiGCN.

Round Partition and Execution. To avoid unacceptable routing latency and redundant off-chip memory accesses, a round execution method coupled with a simple round partition is proposed. Figure 7 illustrates the round partition which is based on a simple graph mapping method to reduce the mapping overhead and simplify packet routing. As illustrated in Figure 7(a), for a vertex, the bits in range $[0, n]$ in the vertex ID (vID) will be the ID of the processing node to which the vertex is partitioned and mapped. The bits in range $[n, x+n]$ in the vID are used to partition and map 2^x vertices with interleaving vIDs into the same processing node together. The bits in range $[x+n, 32]$ in the vID will be the index of round (rID) for simplicity. The value of n is determined by the total number of processing nodes (#total_node) in MultiAccSys, which is equal to $\lfloor \log_2(\text{\#total_node}) \rfloor$. The value of x can be reconfigured for different datasets to better utilize on-chip memory, which is determined by $2^x \leq \frac{\alpha M}{S} < 2^{x+1}$, where M is the capacity of the aggregation buffer and S is the size of aggregated feature vector. The value of α must be less than 1 to reserve space for network communication and round overlap. In our implementation, we set α to 0.75. For each vertex, all its in-coming edges are partitioned into the same round, and used to build the nID list, offset list, and neighbor lists of its all in-coming neighboring vertices for multicast. The compressed sparse row format is used to reorganize the neighbor lists to reduce needs for both memory and network bandwidth. These information of each vertex is mapped into the same processing node with this vertex's feature vector. Note that after the round partition, if a vertex still has too many outgoing neighbors in a round, this packet is further divided into several packets before being sent to network. Figure 7(b) provides an example for round partition with $M = 60$ bytes, $S = 20$ bytes, $x = 1$, and $n = 4$. In this figure, a graph is *first*

Algorithm 3: Round Execution

◁ ① Initialization ▷

- 1 load round info and configure round execution;

◁ ② Load and Send ▷

- 2 **foreach** vertex v in local **do**
- 3 load v 's feature vector, network topology, and graph topology;
- 4 **if** current node has v 's neighbor u **then**
- 5 save a replica in aggregation buffer;
- 6 save {buffer address, list of all neighbor u } to edge buffer;
- 7 **end**
- 8 send v 's data to remote processing nodes;
- 9 **end**

◁ ③ Receive ▷

- 10 receive v 's feature vector and graph topology;
- 11 save a replica in aggregation buffer;
- 12 save {buffer address, v 's neighbors} to edge buffer;

◁ ④ Compute ▷

- 13 perform `aggregate` function using buffer address and neighbors' `vID` in items of edge buffer;
- 14 perform `combine` function when aggregation is complete;
- 15 store final combined result to off-chip memory;

◁ ⑤ Synchronization ▷

- 16 synchronize and complete the current round;

partitioned into two sub-graphs corresponding to two rounds. For example, V_{15} and V_{44} are partitioned into round 0 and round 1 with their in-coming edges, respectively. *Second*, the `vID` of each vertex (e.g., V_{15}) is included into the neighbor lists of its incoming neighbors (e.g., V_0 and V_{54}) to support the scatter-based method. As a result, the large neighbor lists of high out-degree vertices (e.g. V_0) are sliced over several rounds, avoiding large packets.

Algorithm 3 demonstrates the round execution method which includes five steps: ① *Initialization*, where each processing node loads the round info and is configured for a new round. ② *Load and Send*, where each processing node loads graph data including feature vectors, graph topology, and network topology, and then sends the graph data to other nodes. ③ *Receive*, where each processing node receives the replica of feature vector and graph topology from remote nodes into the aggregation buffer and the edge buffer. ④ *Compute*, where each processing node executes the `aggregate` function or `combine` function to process graph data in local or from remote. ⑤ *Synchronization*, where each processing node broadcasts an end signal to others when its workload in the current round is completed, and the current round is terminated after all signals from other nodes are collected. Note that other synchronization mechanisms can also be used for better efficiency. Besides, execution overlap technique is utilized in the round execution to improve resource utilization. After step ①, step ②, ③, and ④ can be overlapped intra round. Moreover, these three steps can also be overlapped inter round. Furthermore, step ④ is able to actively process the graph data locally to keep compute resources busy when no graph data is received.

The round partition and round execution method provide two benefits. *First*, large neighbor lists are sliced, avoiding compute resource underutilization due to the intensive transmission and unacceptable routing latency for a single packet. *Second*, the large volume of replicas are split and processed over a set of rounds, so that replicas in each round can be totally saved in on-chip memory, avoiding the frequent transfer of replicas between on-chip memory

TABLE 2
System parameters of MultiGCN @1GHz & TSMC 12 nm.

Network Parameters			
Network Topology	#Processing Node	Network Bandwidth	Network Latency
2D Torus	16	600 GB/s	500 Cycles
Memory Parameters of Each Processing Node			
Buffer in Router	Buffer in Send Unit	Buffer in Loader	Edge Buffer
1.5 MB	512 KB	896 KB	128 KB
Aggregation Buffer	Weight Buffer	Combination Buffer	HBM Bandwidth
1 MB	2 MB	256 KB	256 GB/s
Compute Parameters of Each Processing Node			
8 Reusable Systolic Arrays (each size 1×128)			

TABLE 3
Graph datasets used in evaluation [28].

Name	$ V $	$ E $	d_v	$ h^0 $	$ h^1 $	Topology Size	Feature Size
Real-world Graphs							
Reddit (RD)	233K	114M	489	602	128	460 MB	561 MB
Orkut (OR)	3M	117M	39	500	128	481 MB	6 GB
LiveJournal (LJ)	5M	69M	14	500	128	295 MB	10 GB
Synthetic Graphs							
RMAT-19 (RM19)	0.5M	16.8M	32	512	128	67 MB	1 GB
RMAT-20 (RM20)	1M	33.6M	32	512	128	134 MB	2 GB
RMAT-21 (RM21)	2.1M	67.1M	32	512	128	269 MB	4 GB
RMAT-22 (RM22)	4.2M	134M	32	512	128	537 MB	8 GB
RMAT-23 (RM23)	8.4M	268M	32	512	128	1074 MB	16 GB

and off-chip memory.

5 EVALUATION METHODOLOGY

Evaluation Tools. We design and implement an in-house simulator to measure execution time in number of cycles. The simulator has a cycle-level model for many microarchitectural components, including multi-bank on-chip buffer, HBM (high bandwidth memory), NVLink, systolic arrays, and so on. To measure critical path delay (in cycles) of the router, receive unit, send unit, loader, scheduler, and compute unit, we implement and synthesize these modules in Verilog. We use the Synopsys Design Compiler with the TSMC 12 nm standard VT library for the synthesis and estimate power consumption using Synopsys PrimeTime PX. The slowest module has a critical path delay of 0.83 ns including the setup and hold time, putting MultiGCN comfortably at 1 GHz clock frequency. The access latency, energy, and area of the on-chip buffer and FIFO are estimated using Synopsys DesignWare Memory Compiler. The access latency and energy of HBM are simulated by Ramulator [24], a cycle-accurate DRAM simulator and estimated with 7 pJ/bit as in [25], respectively. The access latency and energy of NVLink are estimated with around 500 ns as in [26] and 8 pJ/bit as in [27], respectively.

Baselines and System Configurations. To demonstrate the advantages of MultiGCN, we compare MultiGCN with a single-node GCN accelerator (i.e., AWB-GCN [15]) using identical hardware resources, two GPU-based solutions (i.e., PyG [29] and GNNAdvisor [30]) running on T4 GPU, OPPE-based MulAccSys, and OPPE-based MulAccSys. Three configurations of MultiGCN are evaluated to assess MultiGCN. The first configuration is MultiGCN only employing the topology-aware multicast mechanism (TMM), denoted by **MultiGCN-TMM**. The second configuration is MultiGCN only employing the scatter-based round execution mechanism (SREM), denoted by **MultiGCN-SREM**. The last configuration is MultiGCN employing both TMM and SREM, denoted by **MultiGCN-TMM+SREM**. All these configurations use the system parameters described in Table 2.

Workloads. We implement three well-known GCNs in MultiGCN, namely GCN [31], GINConv (GIN) [32], and GraphSAGE (SAG) [33]. Due to the long simulation time on large-scale graphs, we simulate only the first layer of these models. Since runtime

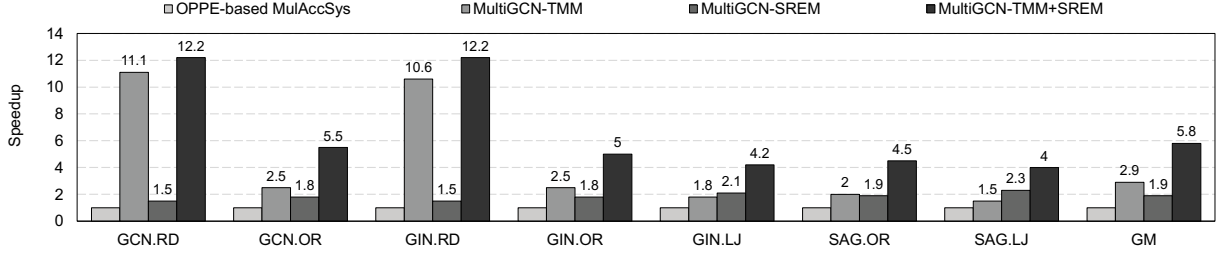
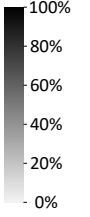


Fig. 8. Performance comparison between OPPE-based MulAccSys and MultiGCN (normalized to OPPE-based MulAccSys).

TABLE 4

Utilization ratio (%) of network bandwidth, DRAM bandwidth, and compute unit of OPPE-based MulAccSys and MultiGCN.

	OPPE-based MulAccSys			MultiGCN-TMM			MultiGCN-SREM			MultiGCN-TMM+SREM		
	Network Bandwidth	DRAM Bandwidth	Compute Unit	Network Bandwidth	DRAM Bandwidth	Compute Unit	Network Bandwidth	DRAM Bandwidth	Compute Unit	Network Bandwidth	DRAM Bandwidth	Compute Unit
GCN.RD	19	12	2	4	29	20	28	17	3	87	14	22
GCN.OR	17	15	6	6	41	16	31	20	12	69	32	35
GCN.LJ	16	20	14	7	41	26	34	23	30	60	31	68
GIN.RD	19	12	2	4	21	21	28	17	3	87	14	24
GIN.OR	17	15	8	6	41	21	31	19	15	62	29	41
GIN.LJ	15	19	19	7	41	34	33	22	40	51	26	80
SAG.RD	19	19	9	8	39	17	45	29	20	76	36	39
SAG.OR	16	18	10	7	42	21	31	21	20	60	31	46
SAG.LJ	15	23	21	8	40	32	34	24	47	53	30	84
GM	17	17	8	6	37	22	33	21	15	66	26	44



characteristics of GCNs are input-dependent, we use several real-world and synthetic graphs as inputs to each GCN model, as shown in Table 3. Topology size of graph refers to the total size of edges, $|E| * 4$ Bytes. Feature size of graph refers to the total size of feature vectors, $|V| * |h^0| * 4$ Bytes.

6 RESULTS

6.1 Overall Results

Performance. Figure 8 compares the performance of the proposed MultiGCN against that of OPPE-based MulAccSys. In this figure, the last set of bars, labeled GM, indicates the geometric mean across all workloads. Our evaluation shows that MultiGCN with only the TMM mechanism or only SREM mechanism outperforms OPPE-based MulAccSys by $2.9\times$ or $1.9\times$ on average. When both mechanisms are employed, MultiGCN achieves $4\sim 12\times$ speedup over OPPE-based MulAccSys, and $5.8\times$ on average.

To provide more insights into the performance improvement of MultiGCN, Table 4 shows the utilization ratios of network bandwidth, DRAM bandwidth, and compute unit in MultiGCN. Compared with OPPE-based MulAccSys, the utilization ratio of network bandwidth, DRAM bandwidth and compute unit of MultiGCN-TMM+SREM improve by $3.88\times$, $1.53\times$, and $7.33\times$ on average, respectively. This points to the main contributor of the large speedup achieved by our design: the TMM mechanism and the SREM mechanism.

Area and Power. Table 5 provides the detailed characteristics of MultiGCN. The area and power of each processing node are 12.4 mm^2 and 3671.13 mW respectively. The buffers including edge buffer, aggregation buffer, weight buffer, and combination buffer occupy most area of the processing node and accounts for 48% power of the processing node. The area and power produced by the compute unit are 6.8% and 17.72%. For the computation precision, we use 32-bit fixed point which is enough for accurate GCN inference. The area and power produced by router are 22.59% and 18.78% due to the large routing buffer and heavy packet transmissions.

Energy and its Breakdown. Figure 9 shows the energy consumption of MultiGCN-TMM+SREM in detail. Figure 9(a) depicts that MultiGCN costs only 28%~68% energy of OPPE-based

TABLE 5
Characteristics of processing node @1GHz & TSMC 12 nm.

Component or Block	Area (mm^2)	%	Power (mW)	%
Processing Node	12.4	100	3671.13	100
Breakdown by Functional Block				
Edge Buffer	0.23	1.88	9.03	0.25
Aggregation Buffer	1.87	15.06	578.3	15.75
Weight Buffer	3.74	30.11	614.13	16.73
Combination Buffer	0.47	3.76	551.42	15.02
Compute Unit	0.84	6.8	650.63	17.72
Router	2.8	22.59	689.45	18.78
Loader	1.52	12.24	320.51	8.73
Send Unit	0.93	7.53	257.6	7.02
Scheduler	4.73E-04	0.00	0.00	0.00
Others	1.89E-03	0.02	0.04	0.00

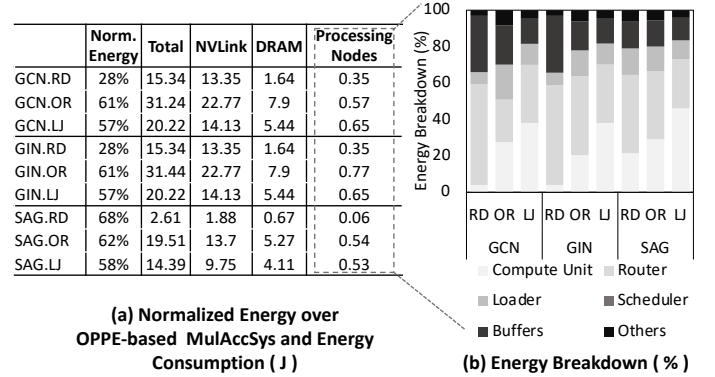


Fig. 9. Energy: (a) Normalized energy over OPPE-based MulAccSys and energy details; (b) Energy breakdown of processing nodes.

MulAccSys. The energy consumed by network (i.e., NVLink) is larger than DRAM and processing nodes in all cases, since the feature vector loaded from DRAM can be shared across multiple remote processing nodes. Figure 9(b) shows that the compute unit, router and buffers consume most energy of the processing nodes.

6.2 Effect and Overhead of Optimizations

To dissect the effect of our optimizations, the normalized network transmission and DRAM access of MultiGCN (normalized to OPPE-based MulAccSys) are shown in Table 6. The reduction

TABLE 6
Normalized network transmission and DRAM access of MultiGCN
(normalized to OPPE-based MulAccSys).

	MultiGCN-TMM		MultiGCN-SREM		MultiGCN-SREM+TMM	
	Trans.	Access	Trans.	Access	Trans.	Access
GCN.RD	2%	21%	100%	93%	37%	10%
GCN.OR	14%	112%	100%	72%	75%	39%
GCN.LJ	25%	114%	100%	53%	79%	33%
GIN.RD	2%	16%	100%	93%	37%	10%
GIN.OR	14%	112%	100%	72%	75%	39%
GIN.LJ	25%	118%	100%	53%	79%	33%
SAG.RD	20%	102%	100%	63%	88%	41%
SAG.OR	22%	116%	100%	60%	81%	38%
SAG.LJ	35%	115%	100%	48%	86%	33%
GM	13%	75%	100%	66%	68%	27%

of redundant transmissions and DRAM accesses as well as the overhead analysis of these optimizations are also shown in Table 7.

Effect. The TMM mechanism helps eliminate the redundant transmissions. Table 6 shows that the network transmission of MultiGCN-TMM is only 13% that of OPPE-based MulAccSys. This is because a single packet containing a replica of the feature vector is sent to many other processing nodes that also request it via multicast. Note that the number of DRAM access in RD dataset decreases, but extra DRAM accesses are introduced in most datasets. This is because a feature vector in RD dataset loaded from DRAM can be shared by many remote processing nodes due to its extremely high average degree (e.g., 489). The SREM mechanism avoids the frequent transfer of replicas between on-chip memory and off-chip memory. Specifically, the rounds are properly partitioned so that the replicas of all vertices and intermediate results in a round always stays on-chip until the computation is done. Compared with OPPE-based MulAccSys, MultiGCN-SREM introduces only 66% number of DRAM accesses on average.

Table 6 shows that when these two mechanisms are employed, both the network transmission and number of DRAM accesses are reduced significantly to only 68% and 27% on average, respectively. Note that the effect of TMM mechanism is hurt by the SREM mechanism because each round may introduce a multicast of the same feature vector. In contrast, TMM mechanism promotes the effect of SREM mechanism since a feature vector loaded from DRAM can be multicast to and shared by many remote processing nodes. Table 7 depicts that MultiGCN-TMM+SREM reduces 32% redundant network transmissions and 100% of redundant DRAM accesses on average compared to OPPE-based MulAccSys.

Overhead. The main optimization overheads are the extras of transmission latency and preprocessing time for round partition, but all of them are small, only 0.21% and 6.1% on average, as shown in the last two columns in Table 7. Note that as more redundant transmissions are reduced, the network topology and graph topology information in the packet increases transmission latency. The round partition accounts for less than 12% time of the graph mapping because it can be coupled into the process of graph mapping. Besides, it is a one-time overhead for each dataset that can be amortized over the execution of different GCN models.

6.3 Comparisons with the State of the Arts

The performance of MultiGCN (1 node) is slightly lower than that of GNNAdvisor running on one T4 GPU, 0.7 \times on average. However, MultiGCN aims to scale single-node accelerator to accelerate GCNs on large-scale graphs efficiently, such as average 3.4 \times speedup of MultiGCN (4 nodes) over GNNAdvisor.

TABLE 7
Reduction of redundant transmission and redundant DRAM access, extra transmission latency, and extra preprocessing time compared with OPPE-based MulAccSys.

	MultiGCN-TMM+SREM			
	Redundant Transmission	Redundant DRAM access	Transmission Latency	Round Partition Time
GCN.RD	-64%	-100%	+0.52%	+6.6%
GCN.OR	-30%	-100%	+0.15%	+12%
GCN.LJ	-30%	-100%	+0.13%	+2.8%
GIN.RD	-64%	-100%	+0.52%	+6.6%
GIN.OR	-30%	-100%	+0.15%	+12%
GIN.LJ	-30%	-100%	+0.13%	+2.8%
SAG.RD	-17%	-100%	+0.07%	+6.6%
SAG.OR	-25%	-100%	+0.11%	+12%
SAG.LJ	-24%	-100%	+0.1%	+2.8%
GM	-32%	-100%	+0.21%	+6.1%

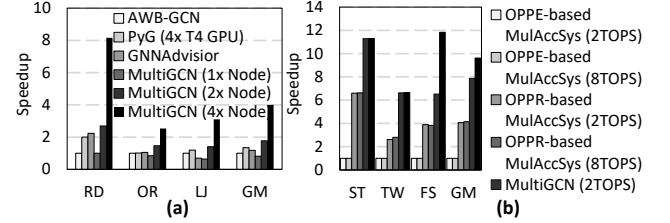


Fig. 10. Comparisons with the state-of-the-arts: (a) Speedup of PyG (4 T4 GPUs), GNNAdvisor (1 T4 GPU), MultiGCN (2 nodes), and MultiGCN (4 nodes) over AWB-GCN; (b) Speedup of MultiGCN (128 nodes) over OPPE-based and OPPE-based MulAccSys (128 nodes).

To demonstrate the advantages of MultiGCN, we compare MultiGCN against the state-of-the-arts. Figure 10(a) shows that the average speedup of MultiGCN (1 node) is slightly lower than that of AWB-GCN and GNNAdvisor. However, MultiGCN aims to scale single-node accelerator to accelerate GCNs on large-scale graphs efficiently, such as on average 4 \times and 3.4 \times speedup of MultiGCN (4 nodes) over PyG with 4 T4 GPUs. This is because GPUs aim at workloads with regular execution pattern, but they cannot efficiently tackle irregular execution patterns of GCNs [11], [15], [16], [34], [35]. Note that GPU performance of PyG is estimated by accumulating the kernel execution time which does not account for the memory copy time and system stack overhead. Besides, PyG leverages mini-batch to make each GPU execute inference independently, so that inter-GPU communications are eliminated. However, mini-batching multiplies data volume in system due to massive copies of neighboring feature vectors in each GPU. Figure 10(b) shows that MultiGCN (128 nodes and 8 TOPS) achieves average 9.6 \times and 2.3 \times speedup over OPPE-based MulAccSys (128 nodes and 8 TOPS) and OPPE-based MulAccSys (128 nodes and 8 TOPS) respectively due to less network transmissions and DRAM accesses. MultiGCN has different speedup on the FS dataset when compute capability increases because the number of network transmissions of FS is little while the number of compute for local data processing is large.

6.4 Exploration of Design

We conduct several experiments on the GCN model to explore our architecture design in terms of hardware sensitivity and graph characteristic sensitivity as follows. Due to the long simulation

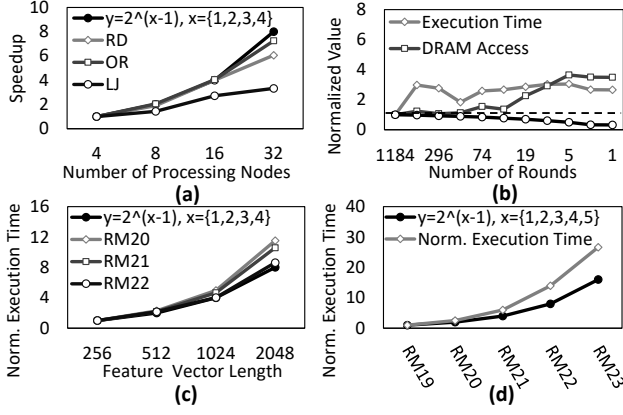


Fig. 11. Hardware sensitivity: (a) Speedup over number of processing nodes; (b) Normalized execution time, normalized amount of network transmission, and normalized amount of DRAM access across number of rounds. Graph characteristics sensitivity: Normalized execution time across (c) feature vector length and (d) scale of vertex number.

time, we simulate the processing for 10% of vertices for each experiment in this subsection.

Hardware Sensitivity. *First*, Figure 11(a) illustrates the speedup across different number of nodes of the 2D torus network topology. MultiGCN gains performance improvement on RD and OR datasets as the number of nodes increases, and the speedup remains linear as the number of nodes increases to 32. Limited by the network bandwidth, the performance gain on LJ dataset gradually decreases as the number of node increases. This is due to the low reusability of feature vector in the network transmission since the average vertex degree of the LJ dataset is low. *Second*, Figure 11(b) shows that the amount of network transmission decreases as the number of round decreases in the processing of LJ dataset because the number of multicast for the same feature vector decreases. Besides, the execution time and number of DRAM accesses are variable across different numbers of rounds. This leaves room for further optimizing the design to fit different requirements, which will be explored in our future work.

Graph Characteristic Sensitivity. *First*, the length of feature vector (i.e., $|h^0|$) doubles in Figure 11(c), which means the amount of workload in both the Aggregation phase and Combination phase double, and network transmission increases even more. However, the execution time increases to more than $2\times$. In particular, the performance is sensitive to the length of the feature vector, which is mainly because the increased network traffic imposes heavier burdens on network. *Second*, the number of vertex in graph doubles in Figure 11(d), which means the amount of workload in both the Aggregation phase and Combination phase and network transmission doubles too. However, the execution time increase to more than $2\times$. That is, the performance is sensitive to the scale of vertex number, since a larger scale of vertex number with the same average degree means higher graph sparsity, which hinders the performance.

7 RELATED WORK

Many software frameworks for GCNs have been developed to relieve programming efforts while improving performance on modern architectures [1], [29], [36], [37], [38]. For example, PyG [29] is built upon PyTorch for easy implementation of graph neural networks (GNNs). It consists of easy-to-use mini-batch loaders for

giant graphs and multi-GPU support. GNNAdvisor [30] propose an adaptive and efficient runtime system for GNN acceleration on GPUs. DGL [36] is a scalable GNN framework that simplifies the development of efficient GNN-based training and inference programs at a large scale in multi-GPU and distributed systems. Unfortunately, the distinct execution pattern in GCNs causes processing inefficiencies on conventional architectures. Therefore, GCNs demand specialized architecture design.

Characterizing the execution pattern and execution semantic of GNNs on GPUs is important for both software and hardware optimizations for GNNs, which has been extensively studied in previous work [12], [39], [40], [41], [42], [43]. Yan et al. [12] reveal the computation and memory accessing pattern of GCNs on GPU. Yan et al. [41] disclose the execution pattern and execution semantic of heterogeneous graph neural networks (HGNNs) on GPU. Zhang et al. [39] focus on understanding the computational graph of GNN, from the perspective of computation, IO and memory. Zhang et al. [40] characterize the computation of a large portion of GNN variants concerning general-purpose and application-specific architectures.

Hardware acceleration for GCNs has been recently explored. Many single-node domain-specific architectures and frameworks have been designed for GCN acceleration [11], [14], [15], [16], [17], [44], [45], [46], [47], [48], [49], [50], [51]. For example, HyGCN [11] proposes a hybrid architecture to address the hybrid execution pattern of GCNs. AWB-GCN [15] targets workload imbalance in the acceleration of GCNs. GCNAX [17] proposes a flexible and optimized dataflow for GCNs that simultaneously improves resource utilization and reduces data transfer. RE-FLIP [52] designs PIM-featured crossbar architectures to build a unified architecture to perform the hybrid execution pattern of GCNs.

The ever-growing scale of graphs has posed new challenges that single-node accelerators cannot sufficiently address. Thus, a multi-node acceleration system is highly desirable. Although a straightforward multi-node design for large-scale GCNs follows Tesseract [20] or other Tesseract-based architectures [21], [53], it suffers from two inefficiencies including a vast of redundant transmissions and off-chip memory accesses. To this end, we propose MultiGCN, an efficient MultiAccSys for large-scale GCNs.

8 CONCLUSION

In this work, we aim to scale the single-node GCN accelerator to accelerate execution of GCNs on large-scale graphs. We first characterize the communication pattern and challenges of multi-node acceleration for GCNs. Guided by our observations, we then propose MultiGCN, an efficient MultiAccSys for large-scale GCNs that trades network latency for network bandwidth. MultiGCN achieves $4\sim 12\times$ and $2.5\sim 8\times$ speedup over baseline MultiAccSys and multi-GPU solution respectively. Designing multi-node acceleration systems is vital to enable practical execution of GNNs on real-world large-scale graphs. We believe our work will draw more attention to the design of domain-specific processor clusters for increasingly important GNNs and graph-structured data.

ACKNOWLEDGMENTS

We sincerely thank Prof. Guang R. Gao for his guidance and contribution to this work. We also would like to express our gratitude to all reviewers' constructive comments for helping

us polish this paper. This work was supported by the National Natural Science Foundation of China (Grant No. 61732018, and 61872335), Austrian-Chinese Cooperative R&D Project (FFG and CAS) (Grant No. 171111KYBS20200002), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-029), Open Research Projects of Zhejiang Lab (NO. 2022PB0AB01), and CAS Project for Youth Innovation Promotion Association.

REFERENCES

- [1] H. Yang, “Aligraph: A comprehensive graph neural network platform,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 3165–3166.
- [2] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, “A comprehensive survey on graph neural networks,” *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.
- [3] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, “Graph neural networks: A review of methods and applications,” *AI Open*, vol. 1, pp. 57–81, 2020.
- [4] Z. Zhang, P. Cui, and W. Zhu, “Deep learning on graphs: A survey,” *IEEE Transactions on Knowledge and Data Engineering*, 2020.
- [5] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, “Geometric deep learning: going beyond euclidean data,” *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.
- [6] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, “Modeling relational data with graph convolutional networks,” in *European Semantic Web Conference*. Springer, 2018, pp. 593–607.
- [7] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, “Graph convolutional neural networks for web-scale recommender systems,” in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 974–983.
- [8] X. Chen, L.-J. Li, L. Fei-Fei, and A. Gupta, “Iterative visual reasoning beyond convolutions,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7239–7248.
- [9] L. Oliver and P. Luis, “Traffic prediction with advanced graph neural networks,” [Online]. Available: <https://deeptmind.com/blog/article/traffic-prediction-with-advanced-graph-neural-networks>
- [10] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan, and B. Yu, “High performance graph convolutional networks with applications in testability analysis,” in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, June 2019, pp. 1–6.
- [11] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, “Hygen: A gen accelerator with hybrid architecture,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 15–29.
- [12] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, “Characterizing and understanding gcns on gpu,” *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 22–25, 2020.
- [13] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “Nvidia tesla: A unified graphics and computing architecture,” *IEEE micro*, vol. 28, no. 2, pp. 39–55, 2008.
- [14] H. Zeng and V. Prasanna, “Graphact: Accelerating gen training on cpu-fpga heterogeneous platforms,” in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 255–265.
- [15] T. Geng, A. Li, R. B. Shi, C. S. Wu, T. Q. Wang, Y. F. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. C. Herboldt, “Awb-gen: A graph convolutional network accelerator with runtime workload rebalancing,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2020.
- [16] S. Liang, Y. Wang, C. Liu, L. He, L. Huawei, D. Xu, and X. Li, “Engn: A high-throughput and energy-efficient accelerator for large graph neural networks,” *IEEE Transactions on Computers*, 2020.
- [17] J. Li, A. Louri, A. Karanth, and R. Bunescu, “Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 775–788.
- [18] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, “Open graph benchmark: Datasets for machine learning on graphs,” *arXiv preprint arXiv:2005.00687*, 2020.
- [19] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, “A domain-specific supercomputer for training deep neural networks,” *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.
- [20] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.
- [21] 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 *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 544–557.
- [22] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, “Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2019.
- [23] M. Li, Q.-A. Zeng, and W.-B. Jone, “Dyxy: A proximity congestion-aware deadlock-free dynamic routing method for network on chip,” in *Proceedings of the 43rd Annual Design Automation Conference*, ser. DAC ’06. New York, NY, USA: Association for Computing Machinery, 2006, p. 849–852. [Online]. Available: <https://doi.org/10.1145/1146909.1147125>
- [24] Y. Kim, W. Yang, and O. Mutlu, “Ramulator: A fast and extensible dram simulator,” *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan. 2016.
- [25] M. O’Connor, “Highlights of the high-bandwidth memory (hbm) standard,” in *Memory Forum Workshop*, 2014.
- [26] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl, “Pump up the volume: Processing large data on gpus with fast interconnects,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1633–1649.
- [27] D. BILL. Gtc china 2020 keynote. [Online]. Available: https://live.nvidia-china.com/20201215-gtc-china-2020/deck-assets/GTC_China_2020_Keynote.pdf
- [28] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” <http://snap.stanford.edu/data>, Jun. 2014.
- [29] M. Fey and J. E. Lenssen, “Fast graph representation learning with PyTorch Geometric,” in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- [30] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, “Gnnadviser: An adaptive and efficient runtime system for {GNN} acceleration on gpus,” in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 515–531.
- [31] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *CoRR*, vol. abs/1609.02907, 2016. [Online]. Available: <http://arxiv.org/abs/1609.02907>
- [32] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?” *CoRR*, vol. abs/1810.00826, 2018. [Online]. Available: <http://arxiv.org/abs/1810.00826>
- [33] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in *Advances in Neural Information Processing Systems 30*, 2017, pp. 1024–1034. [Online]. Available: <http://papers.nips.cc/paper/6703-inductive-representation-learning-on-large-graphs.pdf>
- [34] 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 the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52. New York, NY, USA: ACM, 2019, pp. 615–628. [Online]. Available: <http://doi.acm.org/10.1145/3352460.3358318>
- [35] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, “Graphiconado: A high-performance and energy-efficient accelerator for graph analytics,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.
- [36] Deep graph library. [Online]. Available: <https://docs.dgl.ai>
- [37] J. Thorpe, Y. Qiao, J. Eyolfsson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, S. Fraser, R. Netravali, M. Kim, and G. H. Xu, “Dorylus: affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads,” in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 495–514.
- [38] S. Gandhi and A. P. Iyer, “P3: Distributed deep graph learning at scale,” in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 551–568. [Online]. Available: <https://www.usenix.org/conference/osdi21/presentation/gandhi>
- [39] H. Zhang, Z. Yu, G. Dai, G. Huang, Y. Ding, Y. Xie, and Y. Wang, “Understanding gnn computational graph: A coordinated computation, io, and memory perspective,” *Proceedings of Machine Learning and Systems*, vol. 4, pp. 467–484, 2022.

- [40] Z. Zhang, J. Leng, L. Ma, Y. Miao, C. Li, and M. Guo, "Architectural implications of graph neural networks," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 59–62, 2020.
- [41] M. Yan, M. Zou, X. Yang, W. Li, X. Ye, D. Fan, and Y. Xie, "Characterizing and understanding hgns on gpus," *IEEE Computer Architecture Letters*, pp. 1–4, 2022.
- [42] H. Lin, M. Yan, X. Yang, M. Zou, W. Li, X. Ye, and D. Fan, "Characterizing and understanding distributed gnn training on gpus," *IEEE Computer Architecture Letters*, vol. 21, no. 1, pp. 21–24, 2022.
- [43] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen, "Understanding and bridging the gaps in current GNN performance optimizations," in *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27–March 3, 2021*, J. Lee and E. Petrunk, Eds. ACM, 2021, pp. 119–132. [Online]. Available: <https://doi.org/10.1145/3437801.3441585>
- [44] X. Chen, Y. Wang, X. Xie, X. Hu, A. Basak, L. Liang, M. Yan, L. Deng, Y. Ding, Z. Du, and Y. Xie, "Rubik: A hierarchical architecture for efficient graph neural network training," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2021.
- [45] B. Zhang, R. Kannan, and V. Prasanna, "Boostgcn: A framework for optimizing gcn inference on fpga," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 29–39.
- [46] J. R. Stevens, D. Das, S. Avancha, B. Kaul, and A. Raghunathan, "Gn-generator: A hardware/software framework for accelerating graph neural networks," *arXiv preprint arXiv:2103.10836*, 2021.
- [47] S. Mondal, S. D. Manasi, K. Kunal, and S. S. Sapatnekar, "Gnnie: Gnn inference engine with load-balancing and graph-specific caching," *arXiv preprint arXiv:2105.10554*, 2021.
- [48] Z. Zhou, S. Bizhao, Z. Zhang, G. Yijin, S. Guangyu, and L. Guojie, "Blockgcn: Towards efficient GNN acceleration using block-circulant weight matrices," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021.
- [49] T. Geng, C. Wu, Y. Zhang, C. Tan, C. Xie, H. You, M. Herbordt, Y. Lin, and A. Li, "I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1051–1063.
- [50] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.
- [51] B. Zhang, H. Zeng, and V. Prasanna, "Hardware acceleration of large scale gcn inference," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2020, pp. 61–68.
- [52] Y. Huang, L. Zheng, P. Yao, Q. Wang, X. Liao, H. Jin, and J. Xue, "Accelerating graph convolutional networks using crossbar-based processing-in-memory architectures," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 1029–1042.
- [53] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 712–725.



Gongjian Sun received the B.S. degree from University of Chinese Academy of Sciences, Beijing, China in 2019. He is currently a post-graduate at Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His current research interests include high-throughput computer architecture and graph-based hardware accelerator.



Mingyu Yan received his Ph.D. degree from University of Chinese Academy of Sciences, Beijing, China in 2020. He is currently an Assistant Professor at Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His current research interests include graph-based hardware accelerator and high-throughput computer architecture.



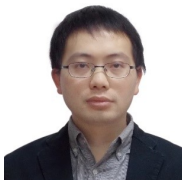
Duo Wang received his B.S. degree from Southeast University, Nanjing, China in 2018. He is currently a Ph.D. candidate at Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His current research interests include high-performance computer architecture and software simulation.



Han Li received the B.S. degree from Jilin University, Changchun, China in 2016. She is currently a Ph.D. candidate at Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. Her current research interests include computer architecture and graph-based hardware accelerator.



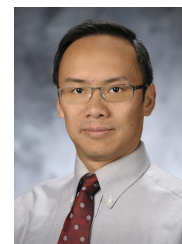
Wenming Li received the Ph.D. degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2016. He is currently an associate professor in Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His main research interests include high-throughput processor architecture, dataflow architecture and software simulation.



Xiaochun Ye received his Ph.D. degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2010. He is currently an associate professor in Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His main research interests include high-performance computer architecture and software simulation.



Dongrui Fan received his Ph.D. degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2005. He is currently a professor and Ph.D. supervisor in Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His main research interests include high-throughput computer architecture and high-performance computer architecture.



Yuan Xie received his Ph.D. degrees from Electrical Engineering Department, Princeton University, Princeton, NJ, USA in 2002. He was a Professor with Pennsylvania State University, State College, PA, USA, from 2003 to 2014. He is currently a Professor with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA, USA.