Accelerating Graph Convolutional Networks Through a PIM-Accelerated Approach

Hai Jin, *Fellow, IEEE*, Dan Chen, Long Zheng, *Member, IEEE*, Yu Huang, Pengcheng Yao, Jin Zhao, Xiaofei Liao, *Member, IEEE*, and Wenbin Jiang, *Member, IEEE*

***Abstract*—*Graph convolutional networks* (GCNs) are promising to enable machine learning on graph data. GCNs show potential vertex-level and intra-vertex parallelism for GPU acceleration, but their irregular memory accesses arising in aggregation operations and the inherent sparsity for vertex features of graphs cause inefficiencies on the GPU. In this paper, we present gPIM, which aims to accelerate GCNs inference through a *processing-in-memory* (PIM) enabled architecture. gPIM is expected to perform compute-intensive combination on the GPU while aggregation and memory-bound combination are offloaded to the PIM-featured *hybrid memory cubes* (HMCs). To maximize the efficiency of such GPU-HMC architecture, gPIM is novel with two key designs: 1) A GCN-induced graph partitioning that minimizes communication overheads between cubes, 2) A programmer-transparent performance estimation mechanism that predicts the performance bound of operations accurately for workload offloading. Experimental results show that gPIM significantly outperforms Intel Xeon E5-2680v3 CPU (8,979.52×), NVIDIA Tesla V100 GPU (96.01×), and a state-of-the-art GCN accelerator AWB-GCN (4.18×).**

***Index Terms*—Accelerators, graph convolutional networks, processing-in-memory.**

I. Introduction

BASED on the great success of deep learning in recent years, neural network over graph data is becoming increasingly important in many real-world applications, such as recommendation [1], node classification [2], and link prediction [3]. *Graph convolutional networks* (GCNs) are exactly neural network models for processing graph data effectively, which have been widely used in the data centers of Google [4], Alibaba [5], and Facebook [6].

GCNs often include two basic kernels for the model inference [7]: *aggregation* and *combination*. For a given vertex over a graph, the aggregation phase gathers the feature vectors from neighbor vertices. This phase heavily relies on the graph structure that is inherently random and sparse. Vertices have different numbers of neighbor vertices, following power-law distribution [8], [9]. Aggregation phase involves a large number of random neighbor vertices accesses; over hundreds of dimensions for vertex feature vector exacerbates the amount of memory access. Based on the gathered features, the combination phase then updates the feature vectors of vertices themselves using a fully-connected *neural network* (NN). In this process, all vertices often share the same reusable weight parameters, and the length of their feature vector is variable (that is, input features and output features may have different lengths).

In the above execution context, GCNs can enjoy the benefits of GPU due to their potential high parallelism and compute-intensive NN-operation [10], [11]. A number of studies [11], [12] focus on improving the locality in graphs and the workload imbalance for GPU threads. However, we identify the bottleneck of GCNs inference on GPUs arising from aggregation operations and occasional combination operations, both of which have a low arithmetic intensity (the number of arithmetic operations per byte [13]). Aggregation operations are always memory-bound due to involving enormous neighbor vertex accesses, while combination operations have uncertain performance bound that depends closely upon the features sparsity and the NN dimension. Even though the modern GPUs (such as Nvidia Tesla P100 and V100) have been equipped with *high bandwidth memory* (HBM), the efficiency of GCNs inference is limited severely because of superfluous data movement between GPU and memory. This slows down the overall performance and increases energy consumption significantly (as discussed in Section II-C). In this paper, we focus on addressing the memory-wall challenges for GCNs inference on GPUs.

We exploit the insight of resolving the aforementioned memory bottleneck by leveraging *processing-in-memory* (PIM) [14]. Micron's *Hybrid Memory Cube* (HMC) [15], stacks several DRAM banks on a logic layer as a cube. And the logic layer is divided into 32 vaults. Each vault can be integrated with digital logic, which can perform computations closely to the memory cells, offering low latency and power consumption benefits. The arithmetic operations are simple for handling aggregation or combination involved in GCNs, such as addition and multiplication. The logic layer can integrate digital logic containing such simple operations under the constraints of the area and power consumption. As demonstrated in previous work [16], HMC cubes are fully compatible with the GPU to constitute

Manuscript received 18 March 2021; revised 22 February 2023; accepted 5 March 2023. Date of publication 15 March 2023; date of current version 9 August 2023. This work was supported by the National Natural Science Foundation of China (NSFC) under Grants 61832006, 62072195, 61825202, and 61929103, and in part by the Huawei Technologies Co., Ltd under Grant YBN2021035018. Recommended for acceptance by C. Li. *(Corresponding author: Dan Chen.)*

The authors are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Clusters and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China (e-mail: hjin@mail.hust.edu.cn; cdhust@hust.edu.cn; longzh@hust.edu.cn; yuh@hust.edu.cn; pcyao@hust.edu.cn; zjin@hust.edu.cn; xfliao@hust.edu.cn; wenbinjiang@hust.edu.cn).

Digital Object Identifier 10.1109/TC.2023.3257514

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

a GPU-HMC architecture, where the traditional GPU memory can be equivalently replaced with the HMC cube(s).

In this paper, we present gPIM, which enables accelerating GCNs by leveraging a GPU-HMC architecture. gPIM is a hybrid computing engine, which incorporates GPU and PIM for both compute-intensive and memory-bound computations. Specifically, gPIM utilizes GPU to process compute-intensive combination while aggregation and memory-bound combination are accelerated by the PIM side. However, achieving such a GPU-PIM architecture for GCNs inference remains challenging.

First, when the accessed data is in a different cube, it causes cross-cube communication with higher latency than the local cube. Neighbor vertices accesses in aggregation may cause serious cross-cube communications for traditional graph partitioning [17]. Although in previous studies [17], [18] they adopted replica to reduce cross-cube communication, it is unpractical for GCNs because long vertex feature vector incurs unaffordable storage overhead. Therefore, a new graph partitioning is needed for gPIM to minimize cross-cube communication. Second, affected by features sparsity and NN dimension, the performance bound of combination is not fixed. Assessing performance bound for combination is critical to determine which computing engine is optimal. Offline profiling can make the best decision, but it introduces expensive costs. It is difficult to choose the preferable computing engine accurately with low overhead.

To effectively accelerate GCNs inference through a PIM-enabled GPU architecture, we propose a software/hardware co-design to exploit the benefit potential of gPIM. First, we observe a new graph partitioning opportunity along feature dimensions and introduce a GCN-induced graph partitioning based on the traditional graph partitioning and this new opportunity, which minimizes cross-cube communication. Second, we observe that a few sampled vertices have similar arithmetic intensity with the whole combination phase. This motivates us to determine computing engine of combination operations by learning arithmetic intensity on a few sampled vertices with low overhead.

In this paper, we make the following contributions:

- We identify the GPU-PIM heterogeneous requirement for the aggregation-combination processing paradigm of GCNs inference, and offload aggregation and memory-bound combination to the PIM side for performance enhancement.
- We present gPIM, which can minimize the communication overheads among HMC cubes with a GCN-induced graph partitioning, and determine the preferable computing engine with a programmer-transparent performance estimation mechanism.
- We evaluate gPIM with a thorough comparison to Intel Xeon E5-2680v3, Nivida Tesla V100, and a state-of-the-art GCN accelerator AWB-GCN, achieving average speedups of 8,979.52×, 96.01×, and 4.18×, respectively.

The rest of this paper is organized as follows. Section II introduces the background and motivation. Section III gives an overview of gPIM. Section IV describes our detail design of gPIM. Section V evaluates gPIM. Section VI discusses the related work and Section VII concludes.

```
1 Procedure GCNsConv()
1   foreach vertex v do
2     edge_ids ← EdgeIndex(v)
3     foreach edgeId ∈ edge_ids do
4       neighbor ← edgeArray[edgeId]
5       v.agg ← Aggregate(v.agg, neighbor.feature)
6   foreach vertex v do
7     v.feature ← Combine(v.agg, weights, biases)
```

Fig. 1. The GCN inference procedure where the aggregation phase is colored in orange and the combination phase is in pink.

II. Background and Motivation

We first introduce some preliminaries of GCNs inference. We then discuss the GPU-PIM heterogeneous requirement and challenges of GCNs inference on a GPU-PIM architecture.

A. GCNs: Programming and Examples

Computational Paradigm. There are many variants of GCNs for different occasions, e.g., GraphSage [19], GIN [20]. Equation (1) and (2) show a layer of a representative model [2] for GCNs. Equation 1 shows the aggregation phase, which aggregates features from all its source neighbors for each vertex. The aggregation phase is used to propagate the information between vertices so that the aggregated information can capture vertex features or the topological information. A and D are adjacent matrix and degree matrix for graph. H represents vertices feature vectors at the k-th layer. F is the aggregated vertices feature vectors. $\tilde{A}$=A+I_N, where I_N is the identity matrix. This adds a self-loop to each vertex in the graph to ensure that the vertex itself features are included when aggregating source neighbor vertices features. $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ is used to normalize A.

Equation (2) shows the combination phase, which updates the feature vector of each vertex to a new one by the weight matrix W. This phase aims to enhance the information representation. Different tasks will focus on capturing different features information.

$$Aggregation : F = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(k-1)} \tag{1}$$

$$Combination : H^k = FW \tag{2}$$

Fig. 1 depicts the general computational procedure of GCNs inference. The codes in orange is the aggregation phase and the pink area is the codes for the combination phase. GCNs applications are easy to write by implementing only two user-defined *Aggregate* and *Combine* functions.

Fig. 2 further depicts a diagram flow to illustrate the GCNs inference. The aggregation gathers the feature vectors from neighbor vertices. It depends on the graph structure to access the vertex features of all source neighbors, and the number of source neighbor vertices among different vertices varies significantly.

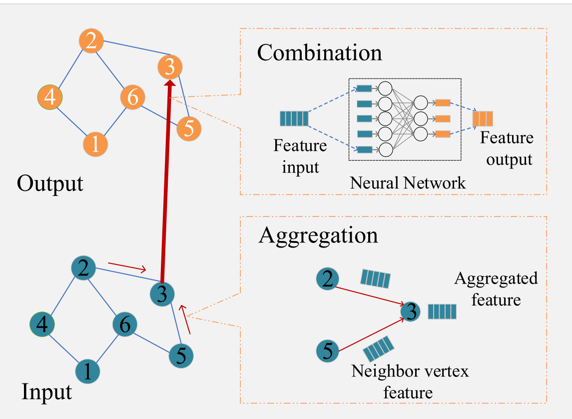

Fig. 2. A diagram flow for the GCN model.

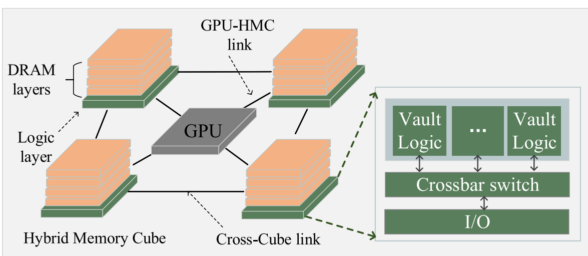

Fig. 3. GPU-HMC architecture.

The combination uses a fully-connected neural network to update the vertex features. Note that the length of vertex feature vector may be changed in this phase, which shows the length of vertex feature vector is variable between layers.

B. GPU-PIM Architecture

Fig. 3 shows a typical GPU-HMC architecture that has been also explored in previous studies [16], [21]. The 3D stacked memory is used to replace the original DDR memory on GPU. Per cube provides up to four serial links for the external interface. The GPU is connected to the 3D stacked HMC cubes, which are also interconnected with each other. The cube is split into 32 vertical slices, called vaults. Each vault contains a memory controller in the logic layer communicating local DRAM banks with *Through-Silicon Vias* (TSVs). The internal bandwidth of each HMC cube can be over 512 GB per second. The external bandwidth between GPU and HMC cubes can be 320 GB per second.

The logic layer of the HMC cubes has digital logic with computing capability. GPU offloads memory-bound operations to the logic layer of HMC to enjoy low memory latency and power consumption. Previous studies indicate that the logic layer is pretty flexible to be integrated with one or more streaming multiprocessors [16] or other customized processing elements [21] for accelerating different types of applications. Therefore, GPU-HMC architecture can often be slightly different for different applications. In this paper, we modify the logic layer of the HMC cubes slightly with customized processing elements to effectively accelerate the inference for GCNs.

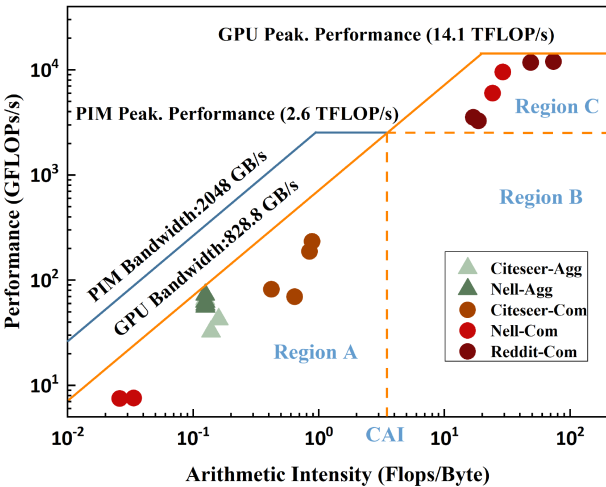

Fig. 4. Roofline of aggregation and combination phases on Nvidia Tesla V100. Agg and Com represent aggregation and combination respectively. E.g., Citeseer-Agg represents aggregation phase for citeseer dataset. The aggregation phase for Reddit dataset runs failure due to out of memory so not shown in the figure.

C. Performance Characterization of GCNs: A Motivating Study

We investigate the performance characteristics of GCNs based on the roofline model [13], which is often used to offer performance estimation of an application on an architecture. Fig. 4 shows the roofline results of benchmarking the GCN model on the COLLAB dataset [22], using the most advanced GCNs framework PyG [23] on Nvidia Tesla V100.

Fig. 4 depicts the data points for the aggregation and combination phases of the roofline model, respectively. We find that aggregation operations have low compute and high memory demand. Aggregation has a relatively fixed arithmetic intensity between datasets and in the memory-bound region, as it aggregates the features of neighbor vertices with performing simple element-wise aggregations. Due to the inherently irregularity of graph structure, aggregation operations involve a large amount of irregular neighbor vertices accesses. In contrast, we observe there is a huge difference in arithmetic intensity for the combination operations. They are distributed in the compute-intensive and memory-bound regions. It often performs vector-matrix multiplication with regular memory accesses. Its arithmetic intensity is dominated by feature vector sparsity and dimension of the NN. Combination operations with extremely sparse features have less compute because there are many zero values in the features. What's more, the small-size NN exacerbates this situation.

Example. Fig. 5 further illustrates an example to facilitate the understanding from the arithmetic intensity perspective. We use ⑥ to reveal the difference between aggregation and combination. Note that we simplify the estimation of arithmetic intensity by using the number of operations per feature.

- *Aggregation*: To aggregate ⑥, its neighbors features (i.e., ②, ⑤, and ①) will be accessed, but their memory accesses are completely random due to the index-sequential storage

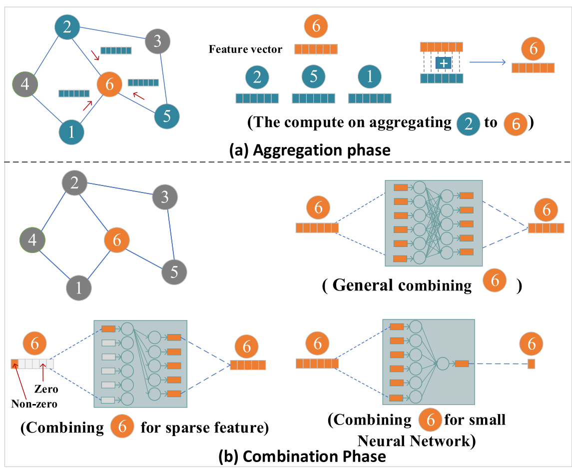

Fig. 5. An example of features access and compute characteristics in GCN model: using ⑥ as an instance on aggregation and combination. ⑥ has high memory access in aggregation while intensive or low compute in combination.

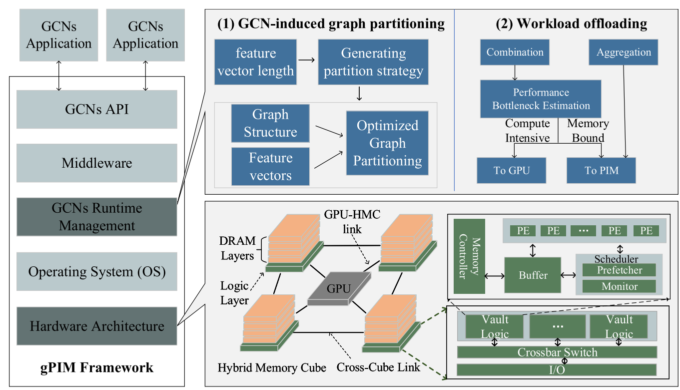

Fig. 6. Overview of gPIM framework.

format. The number of vertex features accesses depends closely on the degree and feature length. Since the computing operation of the aggregation phase is element-wise, the computation times for aggregating a neighbor vertex depend on the feature length. Assume the feature length is 6. Aggregating ⑥ therefore needs 18 operations out of 24 (4×6) feature accesses, yielding the small arithmetic intensity for ⑥ by 0.75.

- *Combination*: To combine a vertex, only its own features are accessed. Hence, its vertex features accesses depend only upon the feature length. There are two factors cooperatively affecting the arithmetic intensity: features sparsity and NN dimension. Let us consider combining ⑥ as an example. Assume the feature length is 6 and the NN is 6×5 fully connected. The computation times for combining ⑥ are 30 (6×5) multiplications and 25 (5×5) additions, yielding the high arithmetic intensity by $\frac{30+25}{6} = 9.16$. However, the features sparsity may lower the arithmetic intensity significantly. Assume only one valid non-zero value in a feature vector. The combination operations contain 5 (1×5) multiplications and 0 (0×5) additions. This will induce the arithmetic intensity as low as 0.8. Further the small-size neural network worsens the situation.

As it is, the compute-intensive combination can be efficiently processed by GPU while aggregation and sparsity-inducing combination are inefficient with memory-bound. This motivates us to accelerate GCNs by leveraging processing-in-memory. To be clear, we divide the region of GPU's roofline into regions A, B, and C, respectively, as shown in Fig. 4. With the optimization of the existing GCNs framework, there are no operations in region B. All the operations are either in region A or C. The operations in region C with high arithmetic intensity can be fast processed by GPU, so these operations should be executed on the GPU. While the operations in region A have low arithmetic intensity with memory-bound, we accelerate these operations by processing-in-memory.

There is a *critical arithmetic intensity* (CAI) that can be used to determine the operations belonging to region A in the Fig. 4. The CAI is the intersection point of upper bounds of the PIM engine and the GPU engine in the roofline model. The upper bound in the roofline model is the peak performance, which is determined by the inherent hardware. Note that regardless of what PIM technology is used, the roofline model of GPU-PIM architecture is similar to Fig. 4. This is because GPU's compute capability is always higher than the PIM side, and memory bandwidth of the PIM side is larger than the GPU. Hence, we can establish a representative formula based on the roofline model. At the intersection point, PIM's peak performance (P_{PIM}) is equal to the GPU's peak bandwidth (B_{GPU}) multiplied by the arithmetic intensity (i.e., CAI). Thus we can obtain a representative formula as shown in Equation (3). For any PIM technology, we can obtain the CAI value based on the peak performance of the PIM side and the peak bandwidth of the GPU. Then the operation with arithmetic intensity lower than CAI could be offloaded to the PIM side, while other operations remain in the GPU for execution.

$$CAI = \frac{P_{PIM}}{B_{GPU}}. \tag{3}$$

Nevertheless, exploiting the GPU-PIM architecture for GCNs remains challenging. First, severe cross-cube communications exist for the aggregation phase due to the neighbor vertices in the different cubes. The long vertex features further exacerbate the cross-cube communication. Second, accurately estimating the performance bound in the combination phase is challenging. The arithmetic intensity in this phase depends not only on the sparsity of features but also on the NN dimension. To solve these challenges, we propose gPIM to accelerate inference of GCNs.

III. gPIM Overview

Fig. 6 shows an overview of the gPIM framework. gPIM enables processing-in-memory for aggregation and memory-bound combination, which requires changes in hardware and software. These changes are transparent to the programmer because the GCNs application layer is separate from others. Programmers can use the same interfaces provided by the existing GCNs library to write GCNs applications as usual. We first give

a hardware architecture overview of gPIM and then describe our design in GCNs runtime management as follows.

A. Hardware Architecture

To enable processing-in-memory, we select the hybrid memory cube of the emerging 3D stacking technology, stacking multiple DRAM banks on a logic layer as a cube. The whole architecture is configured with a GPU connecting to multiple HMC cubes. gPIM follows the HMC Gen2 specifications [24] with 32 vaults for each cube. Crossbar switch is used cross vault communications. Each vault in the logic layer consists of memory controller, scheduler, buffer, and PE.

Memory Controller. The key role of memory controller is to send memory requests from logic layer to DRAM banks. Each vault can send requests to the local memory controller to fetch data from its responsible DRAM die. It also can request data to the remote memory controller through crossbar switch. The crossbar switch is connected with I/O, which is the interface that connects with the host and other HMC.

Scheduler. To hide the latency of memory accesses and exploit the high memory bandwidth, a prefetcher is contained in the scheduler. After edge lists are loaded, we can accurately know the vertices that need to perform subsequently. The prefetcher can prefetch the feature vector of vertices to be accessed into the buffer in advance. In this way, the memory access latency can be overlapped by the prefetching scheme. The scheduler also includes a monitor to support programmer-transparent performance estimation mechanism (discussed in Section IV-B). It monitors the non-zero value in features of sampled vertices and the number of memory transactions.

Buffer. The buffer stores prefetched vertices features and shared weight parameters of NN. For aggregation, the buffer stores neighbor vertices features to be aggregated next time. For combination, the buffer stores the weight parameters of NN, which are shared by all vertices and have high reusability. The weight parameters may exceed the storage space of the buffer. To adapt the buffer size, we slice the weight parameters into multiple sub-weight parameters to complete combination.

PE. We integrate multiple *processing elements* (PEs) inside the logic layer for each vault. Each PE consists of an adder and a multiplier with 32-bit floating-point. The execution mode of PEs can be parallelized not only from the vertex level but also from the feature dimensions. The vertex level granularity is coarse-grained, where each PE is responsible for the workload of a vertex. This mode needs a large buffer to load feature vectors, incurring unaffordable space overheads. We present to parallelize PEs from the feature dimensions. Specifically, we feed fixed-size features to each PE, e.g., 8 features per PE. If all features of a vertex cannot fill all the PEs, the next vertex features will be assigned to the idle PEs. In this way, all the PEs can keep busy without extra space overheads.

B. GCNs Runtime Management

To efficiently accelerate GCNs inference in a GPU-PIM architecture, we offer the following key designs.

GCN-Induced Graph Partitioning. Unlike traditional vertex-centric and edge-centric partitioning in graph processing [25], each vertex in GCNs has a long-dimension feature vector. In this case, aggregating neighbor vertices across different cubes may introduce serious cross-cube communication. We observe a new graph partitioning opportunity to reduce cross-cube communication that divides the features with the same dimension for all vertices into the same cube. To prevent the excessive partitioning from the feature dimensions destroying the feature's locality, we introduce a GCN-induced graph partitioning based on the traditional graph partitioning and this new graph partitioning opportunity for GCNs (discussed in Section IV-A).

Workload Offloading. The combination workloads can be either compute-intensive or memory-bound. Running them on the GPU or the PIM is dependent on performance bottleneck. We propose a programmer-transparent performance estimation mechanism that can automatically evaluate the performance bottleneck of the combination phase. The basic idea behind this mechanism lies in an observation that combination performing behavior for each vertex is similar. gPIM leverages this insight to predict the performance bottleneck before the normal execution with negligible overhead (discussed in Section IV-B).

IV. Design

We now discuss what graph partition to use and how to accurately offload the preferable operations to maximize the efficiency of a GPU-PIM architecture.

A. GCN-Induced Graph Partitioning

Source-based and destination-based partitioning methods are usually used in traditional graph processing [17], [18], [26]. Source-based partitioning causes both intra-cube synchronization and cross-cube communication to co-exist on the same destination vertex. While destination-based partitioning allows us to decouple the intra-cube synchronization on the destination vertices and cross-cube communication to the source vertices. This enables solving cross-cube communication problems independently [26], simplifying the design. In this paper, we focus on destination-based partitioning, where all vertices are distributed to each cube and all the incoming edges with the same destination vertex are also allocated to the same cube, as shown in Fig. 7(a). Source and destination vertices of the grids on the diagonal are located in the same cube. Other grids on the non-diagonal involve cross-cube communication because source and destination vertices are distributed on the different cubes. Suppose that the edges are evenly distributed in the graph; destination-based partitioning has a lot of cross-cube remote accesses. Previous work [17] uses vertex replica to reduce cross-cube communication for traditional graph processing, i.e., a cube can generate a replica of source vertex if this cube only has its destination vertex and the edge. However, directly using replica is impossible for GCNs. The extremely long vertex feature vector brings unaffordable replica storage overhead.

Considering the independent feature dimensions and the aggregation are element-wise, we can aggregate the feature vector

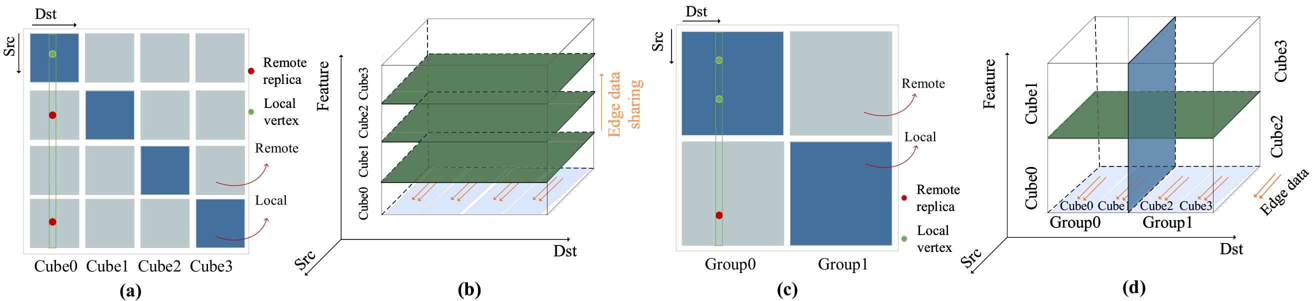

Fig. 7. (a) Traditional graph partitioning for destination-based with replica, (b) Feature-dimensions partitioning, (c) Destination-based partitioning with replica between groups, and (d) Feature-dimensions partitioning within group.

separately in the aggregation phase. This provides a new partitioning opportunity to reduce cross-cube communication, which we call feature-dimensions partitioning. It divides the features with the same dimension for all vertices into the same cube as shown in Fig. 7(b). For instance, assuming vertex features have 400 dimensions, and we divide all the vertex features of 1-100 dimensions into the same cube. In this partitioning, the feature aggregation is completed at the local cube without cross-cube communication. However, there are disadvantages of applying this partitioning directly. When the vertex features are relatively short, partitioning along the feature dimensions can destroy the spatial locality. In addition, each cube completes only part of the features aggregation, which means that each cube needs to access the edge data multiple times.

In order to reduce cross-cube communication and protect the spatial locality of features, we have the following insights. If it is sufficient to divide the feature dimension to all cube while the feature's spatial locality can be preserved, feature-dimensions partitioning will be used. If the feature dimensions are divided into only a limited number of cubes (instead of all cubes), feature-dimensions partitioning and destination-based with vertex replica are used together. In a special case, if the feature vector length is too small to partition, gPIM uses destination-based partitioning with vertex replica to reduce the cross-cube communication. In this paper, we introduce a GCN-induced graph partitioning algorithm to model this insight.

GCN-Induced Graph Partitioning Algorithm. We partition a graph as follows.

1) We define groups, where a group can contain multiple cubes. All cubes will be divided into groups. We stipulate that feature-dimensions partitioning is only used between cubes within the group, while destination-based partitioning with vertex replica is employed between groups.
2) The number of cubes in a group depends on the length of features. Note that we keep the partitioned features at least memory access granularity to protect the spatial locality. We can obtain the number of cubes in a group according to the length of the vertices feature as follows.

Assume the number of total cubes No_c is the power of 2. Memory access granularity is denoted as B_m bytes. With the feature vector length L_f and per feature stored by B_f bytes for a graph, we can easily get the number of cubes for each group (NoG_c) and the number of groups (No_g) by:

$$No_g = \frac{No_c}{NoG_c}, NoG_c = \begin{cases} 2^{\lfloor \log_2 \frac{L_f B_f}{B_m} \rfloor} & \text{if } \frac{L_f B_f}{B_m} < No_c \\ N_c & otherwise \end{cases} \tag{4}$$

$\frac{L_f B_f}{B_m}$ is the maximum number of cubes that the vertex feature vector can be divided into from feature dimensions. $2^{\lfloor \log_2 \frac{L_f B_f}{B_m} \rfloor}$ guarantees that NoG_c is the power of 2 so that each group has the same number of cubes. In this partitioning algorithm, if the number of cubes in a group is the total number of cubes, only feature-dimension partitioning is used. If the number of cubes in a group is 1, only the destination-based partitioning with replica is used. If otherwise, both are used together.

Fig. 7(c) and (d) further depict an example with two groups and two cubes per group to illustrate graph partitioning algorithm. Fig. 7(c) shows that destination-based partitioning between groups. We partition all vertices evenly distributed to each group. All the incoming edges with the same destination vertex are allocated to the same group. We use replica to eliminate cross-group remote accesses. A group can generate a replica of source vertex if this group only has its destination vertex and the edge.

Fig. 7(d) shows the feature-dimensions partitioning within the group. We discuss feature data and edge data placement, respectively. For feature data, we partition from the feature dimensions evenly to all cubes within a group. Thus the accesses to neighbor vertex features for each cube are local. For edge data, we cut NoG_c (the number of cubes within a group) portions along the direction of destination and all incoming edges for each portion are assigned to a cube. Although graph partitioning may be diverse with variable vertex feature length between layers for GCNs, edge data can keep staying in the original cube because it is essentially divided equally among cubes. Each cube in the group needs to access the same edge data. We further introduce broadcasting edge data to reduce DRAM access.

Replica synchronization is necessary when the vertex features update (this happens in the combination phase). To hide the replica synchronization overhead, when a master vertex has completed its features transformation, it will immediately

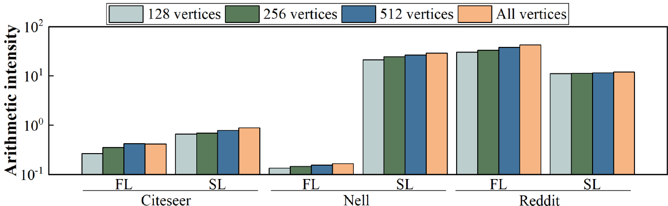

Fig. 8. The arithmetic intensity for the combination phase of a two-layers GCN model with different numbers of sampled vertices, 128, 256, 512, and all respectively. FL and SL are the first layer and the second layer respectively for the GCN model.

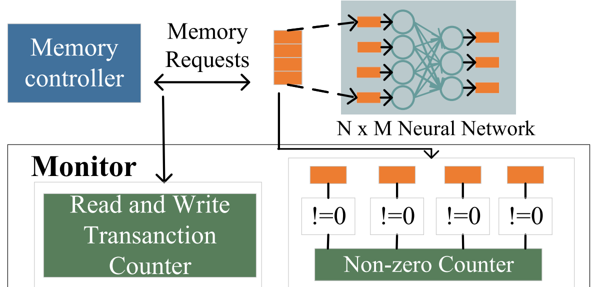

Fig. 9. An example of learning arithmetic intensity on a monitor.

perform its replica synchronization. In this way, the replica synchronization overhead can be overlapped with the combination phase.

B. Workload Offloading

To determine which compute engine (GPU or HMC) is preferable to process combination kernel, one intuitive approach is to use pre-processing, but it introduces too much pre-processing overhead. To address this problem, we first analyze the combination phase behavior to observe opportunities. Based on our observations, we propose a programmer-transparent performance estimation mechanism to predict the optimal engine before the combination phase execution.

Fig. 8 depicts the arithmetic intensity for the combination phase of a two-layers GCN model with different numbers of sampled vertices, 128, 256, 512, and all, respectively. This implies that the arithmetic intensity of a few sampled vertices is close to all vertices of the whole combination phase. It is intuitive because each vertex feature vector transforming has similar computation and memory access behavior. This observation motivates us to propose a programmer-transparent performance estimation mechanism, which learns arithmetic intensity by considering only a few sample vertices and then applies this arithmetic intensity to determine the best compute engine.

The arithmetic intensity is correlated with the number of arithmetic operations and memory transactions in bytes. For arithmetic operations, instead of detecting the number of processing element calculations, we simplify the collection way. Assuming sampled vertices complete combination phase using a $N \times M$ fully-connected neural network, we multiply the number of non-zero value No_{n_z} in sampled vertices features by M to get the number of multiplications, and then multiply by 2 (the approximate number of additions) to get the final number of arithmetic operations. For memory transactions, we track the read transactions T_r and write transactions T_w to memory, and then multiply the transaction size T_s. With a critical arithmetic intensity CAI (discussed in Section II-C), we can easily determine where this combination phase is expected to be executed by:

$$WhichEngine = \begin{cases} PIM & \text{if } \frac{2No_{n_z}M}{(T_r+T_w)T_s} < CAI \\ GPU & otherwise \end{cases} \quad (5)$$

However, at present, GPU does not support collecting arithmetic intensity at runtime. To analyze the arithmetic intensity of sampled vertices, we support it on the PIM side. We integrate a monitor to each vault in the logic layer. Fig. 9 shows an example of learning arithmetic intensity on a monitor, which is area-efficiently implemented with a few OR gates and two counters. It monitors the non-zero value for sampled vertices features and the number of memory transactions. After learning arithmetic intensity, we compare it with CAI. If the learned arithmetic intensity exceeds the CAI, the performance bottleneck is compute-intensive and this combination should execute on GPU. Otherwise, this combination is memory-bound and is accelerated by PIM side. In the case where the combination phase is offloaded to the PIM side, since the vertex features may be partitioned in multiple cubes due to our proposed graph partitioning, the cube completes the computation of the combination for its assigned features and generates intermediate results. These intermediate results in different cubes are finally merged.

Note that the preferable compute engine is selected by aggregating the results of all vaults in all cubes, instead of any single vault. A single vault only performs several vertices instead of all sampled vertices; if these vertices are outlier vertices in the sampled vertices, this will result in a different trend in the arithmetic intensity observed in this vault from others, and thus cause an incorrect selection for the compute engine. Therefore, we use the aggregated results from all vault (in all cubes) to compare with CAI.

GPU and PIM-enabled cubes also exist idle resources when one side performs assigned task. We make full use of computing resources by allocating part of the task to the other one. Meanwhile, each cube performs the assigned vertices that involves different workloads to be processed, which results in potential workload imbalance. For vertices, they may have different number of neighbor vertices to aggregate in the aggregation phase and non-zero value to combine in the combination phase. To reduce workload imbalance, we introduce a stealing mechanism that free cube fetches workloads from busy cube.

V. Evaluation

A. Experimental Setup

gPIM contains host processor and PIM engine. For the host processor, we use Nvidia Tesla V100. And we employ pyG [23], a state-of-the-art GCNs framework that consists of various GCN

TABLE I
PLATFORM SPECIFICATIONS

<table>
<tr><td colspan="3">Host (Tesla V100 SXM2)</td></tr>
<tr><td>GPU</td><td colspan="2">Shading Units 5120 @ 1290 MHz; L1 128 KB (per SM), L2 6 MB; Main Memory 32 GB HBM2</td></tr>
<tr><td colspan="3">PIM</td></tr>
<tr><td>HMC</td><td colspan="2">32GB, 4 cubes, 32 vaults per cube, 312.5MHz</td></tr>
<tr><td rowspan="3">Component in vault</td><td>PE</td><td>including a multiplier and an adder with 32-bit floating-point, multiplier 6 ns latency, adder 2ns latency; 0.0041 W, 0.0053 mm^2</td></tr>
<tr><td>Buffer</td><td>4 KB, 1-cycle latency, 0.0464 W, 0.139 mm^2</td></tr>
<tr><td>Scheduler</td><td>2-cycle latency, 0.00046 W, 0.0055 mm^2</td></tr>
<tr><td>Crossbar</td><td colspan="2">6-cycle all-to-all crossbar in the logic layer [30] layer [36], 2 pJ/bit internal, 8 pJ/bit logic layer [31], [32]</td></tr>
<tr><td>Serials links</td><td colspan="2">10-cycle latency, including 3.2 ns for SerDes, 2pJ/bit [33]</td></tr>
<tr><td>3D DRAM timings</td><td colspan="2">t_{CK}=1.6 ns, t_{CAS}=11.2 ns, t_{RCD}=11.2 ns, t_{RAS}=22.4 ns, t_{RP}=11.2 ns, t_{WR}=14.4 ns</td></tr>
</table>

TABLE II
EVALUATED GCNS DATASETS

Datasets	#Vertices	#Edges	Feature Length
Cora (CR)	2,708	10,556	1433
Citeseer (CS)	3,327	9,104	3703
Pubmed (PB)	19,717	88,648	500
Nell (NL)	65755	251,550	5415
Reddit (RD)	232965	114,615,892	602

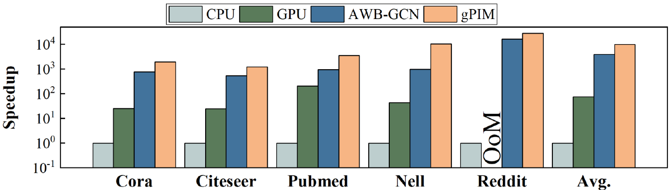

Fig. 10. Speedup of gPIM compared with CPU, the state-of-the-art GPU and AWB-GCN.

models for deep learning on graphs, to execute the GCNs on the host processor. The execution time and energy consumption of the host are obtained by NVProfiler [27] and Nvidia-smi [28], respectively. For the PIM engine, we have implemented a cycle-accurate simulator based on HMC-sim3.0 [15]. We use Cadence Innovus [29] to obtain the execution latency, energy consumption, and area of intra-vault design. Finally, we combine the results from both the host processor and our modification PIM simulator.

gPIM Configurations. Table I shows the specific configuration for gPIM. The host is Nvidia Tesla V100, which contains an L1 cache with 128 KB for each SM, a shared L2 cache with 6 MB, and 32 GB HBM2 with 828.8 GB/s of bandwidth. The PIM side is configured with 4 cubes following the HMC Gen2 specifications [24], and its frequency is set at 312.5 MHz. The memory capacity of each cube is 8 GB and has a total of 32 GB for gPIM. The cube provides 512 GB/s of the internal memory bandwidth to the logic layer and 320 GB/s of external memory bandwidth to the external links [24], [26]. The logic layer for each cube contains 32 vaults. All vaults are connected through a crossbar switch with routing latency of 6 cycles [30]. Each vault contains 32 processing elements and 4 KB SRAM buffer.

Methodology. We compare gPIM with three state-of-the-art designs on typical platforms (1) **Baseline:** Intel Xeon E5-2680v3 CPU; (2) The Nvidia Tesla V100 GPU; (3) A prior art GCN accelerator - AWB-GCN [8]. For fair comparison, we use the state-of-the-art GCNs framework PyG [23] on CPU and GPU. PyTorch Profiler [34] is used to obtain the execution time and the energy consumption is estimated by Intel Product Specifications [35] for the CPU. While the execution time and energy consumption of GPU are collected by NVProfiler [27] and Nvidia-smi [28], respectively. AWB-GCN is an autotuning workload balancing accelerator to accelerate GCN inference on the FPGA platform. We configure the total number of PEs of AWB-GCN with 4096.

GCN Workloads. Table II shows the five datasets used in our evaluation from COLLAB dataset [22]. These datasets are widely used in previous GCN research [7], [8]. We evaluate five datasets with the GCN model [2] containing two layers. In order to compare other platforms fairly, we customize 128 to the feature vector length for the hidden layer, which is the same as the previous works [7], [8].

B. Overall Performance

Fig. 10 shows the performance results of gPIM against CPU, GPU, and AWB-GCN [8].

1) gPIM versus CPU and GPU: gPIM outperforms CPU average 8,979.52×. Compared with the state-of-art Nvidia Tesla V100, gPIM is faster by 96.01× on average. This is because we exploit the advantage of processing-in-memory to overcome the memory-bound inefficiencies in GCNs inference. From Fig. 10, we can see that reddit dataset failed to run on GPU due to being out of memory. PyG framework leverages scatter and matrix multiplication optimization functions to perform GCN [23]. Although this way is good for improving performance, a lot of storage space is needed for data copy. In contrast, gPIM directly aggregates the neighbor vertices in memory.
2) gPIM versus AWB-GCN: AWB-GCN is a state-of-the-art GCN accelerator with runtime workload rebalancing. Although the substantial efforts are used in AWB-GCN, gPIM provides a better speedup of 4.18× on average compared with AWB-GCN. The reason behind this is that gPIM leverages a hybrid engine for both compute-intensive and memory-bound computations. gPIM not only enjoys lower memory latency and high memory bandwidth (up to 512 GB/s) from 3D stack memory but also utilizes the excellent parallel computing capabilities of GPU. Although AWB-GCN also exploits data reuse with software optimization to reduce off-chip memory access, they still have to access the off-chip memory, which has limited memory bandwidth.

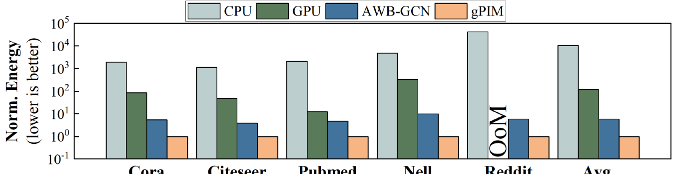

Fig. 11. Energy consumptions of gPIM compared with CPU, the state-of-the-art GPU and AWB-GCN. OoM represents running failure due to out of memory.

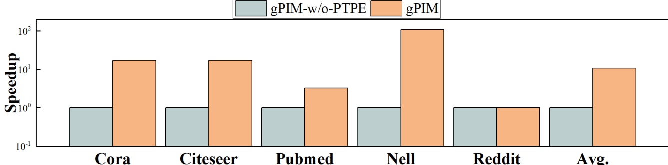

Fig. 13. Performance results of gPIM and gPIM without programmer-transparent performance estimation.

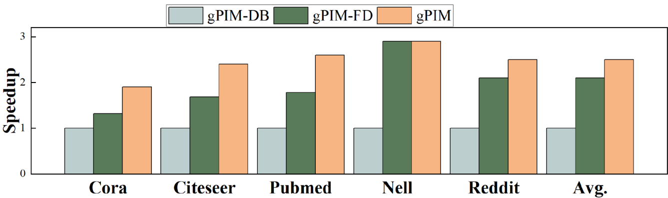

Fig. 12. Performance results of aggregation phase with destination-based, feature-dimensions, and GCN-induced graph partitioning respectively.

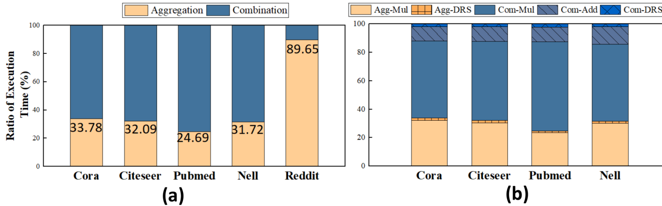

Fig. 14. The breakdown of (a) execution time for the aggregation and combination and (b) latency considering the adder and multiplier. DRS indicates data request stall.

C. Energy Consumption

Fig. 11 describes the energy consumption. gPIM outperforms CPU average of 10,583.79×. Compared with GPU, gPIM's improvements are 119.85× on average. Also, gPIM is superior to AWB-GCN with 5.97× less energy consumption on average. There are two main reasons. First, gPIM benefits from the inherent advantages of PIM. Computation is performed near the data stored, which can greatly reduce the movement of data on the bus. Second, our GCN-induced graph partitioning further reduces energy consumption. It ensures that all neighbor vertices features are accessed locally, and avoids the high energy consumption caused by cross-cube communication. Although edge data needs to be accessed by all cubes in the group, gPIM saves energy by broadcasting the edge data to reduce DRAM access.

D. Performance Breakdown of gPIM

To better understand the effectiveness of our design, we have evaluated the following three aspects: GCN-induced graph partitioning, programmer-transparent performance estimation mechanism, and the latency breakdown of involved operations.

1) GCN-Induced Graph Partitioning: Fig. 12 describes the performance results of the aggregation phase with destination-based (gPIM-DB), feature-dimensions (gPIM-FD), and our GCN-induced graph partitioning (gPIM), respectively. Compared with the baseline of gPIM-DB, the performance improvements of gPIM are 2.51× on average. This is because our graph partitioning enables all features of neighbor vertices to be local, thus reducing most of the cross-cube memory accesses. gPIM outperforms gPIM-FD 1.26× on average. Excessive partitioning over feature dimensions damages the spatial locality of features, which reduces the data fetching efficiency (that is, more times of memory accesses are required for the same features). Our GCN-induced graph partitioning not only eliminates cross-cube memory accesses of features but also protects the spatial locality of features. gPIM-FD and gPIM have the same performance on the nell dataset. This is because nell's feature vector length is always long between layers, and GCN-induced graph partitioning evolves into feature dimensions partitioning.

2) Programmer-Transparent Performance Estimation: We have evaluated the performance results of gPIM and gPIM without programmer-transparent performance estimation (gPIM-w/o-PTPE) as shown in Fig. 13. Although there exists runtime overhead for learning arithmetic intensity, it only accounts for 1.4% of the total running time on average (evaluated in Section V-G). Compared to gPIM-w/o-PTPE, gPIM is faster 10.80× on average. This benefits from the accurate estimation of the performance bottleneck for combination operations and determining the preferable compute engine to accelerate it. Nell can obtain relatively large benefits because this dataset has extremely sparse and long features, and the combination phase takes up more time. This demonstrates that the effectiveness of our programmer-transparent performance estimation is also sensitive to the time taken by the combination operations. For reddit dataset, its aggregation operations almost take up the main time and its features are also density. Our programmer-transparent performance estimation has no performance improvements on reddit dataset.

3) Latency Breakdown for Involved Operations: Fig. 14(a) provides a breakdown of the aggregation and combination execution time ratio in the GCN model. This well demonstrates the execution time proportion of the two phases in gPIM. Fig. 14(b) further shows the latency breakdown considering the adder and multiplier. To facilitate experimental data collection, we only collect some datasets containing a layer whose aggregation and combination are subject to memory-bound and offloaded to the PIM side. Since the multiplier and adder are pipelined, the adder latency can be covered by the more complex multiplier latency. For aggregation, since normalized optimization is used

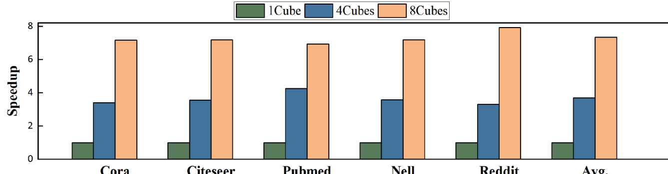

Fig. 15. The overall performance of gPIM as the number of HMC cube increases.

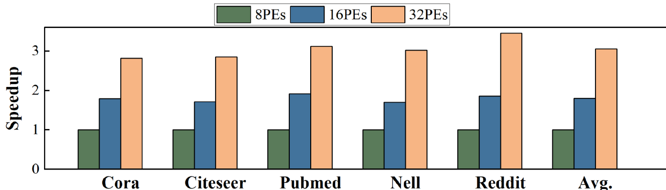

Fig. 16. The overall performance of gPIM as the number of PE in each vault increases.

for the adjacency matrix, neighbor vertices features are multiplied by their influence weights, and then feature aggregation is performed, so the latency of this phase is dominated by the multiplier. For combination, because the vertex features may be partitioned into multiple cubes due to our proposed graph partitioning, each cube completes the computation of the combination for its assigned features and generates partial results, which is dominated by the multiplier latency. These partial results are finally merged, which is dominated by the adder latency.

E. Scalability

We also evaluate the performance of gPIM with the varying number of cubes and intra-vault PEs.

#Cube: Fig. 15 shows the performance of gPIM with increasing the number of cubes. As the number of cubes increases from 1 to 4, the performance has improved by 3.69× on average. Increasing the number of cubes from 4 to 8 improves the performance by 7.64×. The reason behind this is the existence of high-degree vertex features parallelism of GCNs and our data placement strategy; increasing the number of cubes not only makes good use of parallelism but also does not generate much cross-cubes communication. This shows that gPIM has good scalability.

#PE: Fig. 16 shows the performance results of our gPIM with the number of PEs increases. As the number of PEs increases from 8 to 16, the performance has improved by 1.79× on average. Increasing PEs from 16 to 32 improves the performance by 1.69×. The workload can be distributed along vertices and vertex features, which can generate sufficient parallelism to make all PEs busy. However, as the number of PEs continues increasing, we can see the memory bandwidth utilization of the intra-cube has reached its peak from Fig. 17. Therefore, the number of PEs as 32 appears to be the best for gPIM.

 # GCN-induced Graph Partitioning: Our graph partitioning also benefits multiple GPUs scenarios. Fig. 18 describes the speedups of our graph partitioning compared with destination-based partitioning on 4 GPUs. We select three large datasets that are usually used to perform GCN models between multiple GPUs [10]. Compared with destination-based partitioning, the speedup of GCN-induced graph partitioning is 3.13× on average. This is because the GCN-induced graph partitioning is designed to reduce the data movement overhead caused by a large number of random neighbor vertex accesses. It also works with multiple GPUs.

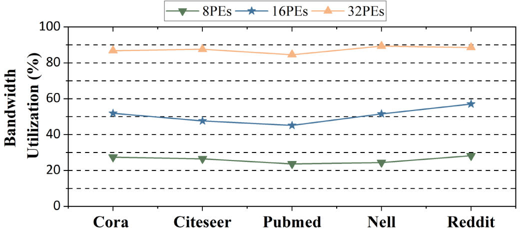

Fig. 17. Bandwidth utilization of 8PEs, 16PEs, and 32PEs.

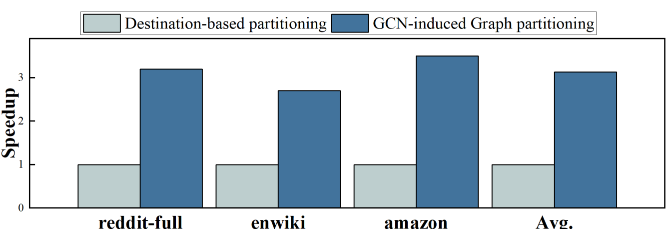

Fig. 18. Speedup of our GCN-induced graph partitioning compared with destination-based partitioning on 4 GPUs.

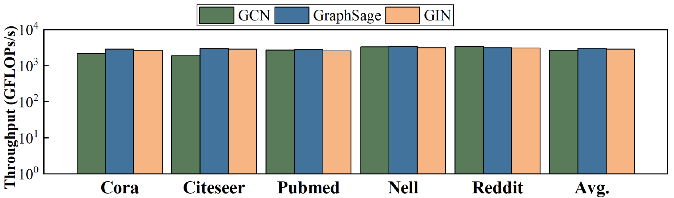

Fig. 19. Throughput of GCN, GraphSage, and GIN.

F. Generality

Fig. 19 shows the throughput of gPIM with GCN and its variants. GraphSage [19] is a GCN variant that uses an inductive learning framework that can generate embedding for unknown vertices and samples neighbors of each vertex. GIN [20] is a model to distinguish if a graph is isomorphic. It proves that Weisfeiler-Lehman is the upper limit of the power of graph neural networks. From Fig. 19, we observe that gPIM can provide similar throughput for GCN variants. This shows that gPIM also has good generality in GCN variants. On the contrary, AWB-GCN [8] can only be used for GCN acceleration.

TABLE III
THE PERCENTAGE OF RUNTIME OVERHEAD FOR GPIM

	Cora	Citeseer	Pubmed	Nell	Reddit	Avg.
Overhead	3.63%	2.94%	0.46%	0.11%	0.01%	1.43%

G. Overhead Analysis

We also discuss the overhead of gPIM in terms of its three aspects: runtime, area, and thermal overhead.

Runtime Analysis: Table III shows the percentages of runtime overhead for learning the arithmetic intensity of our programmer-transparent performance estimation mechanism. In our evaluation, we use 128 sampled vertices to learn. The learning arithmetic intensity only causes 1.43% runtime overhead on average. This demonstrates our mechanism brings substantial performance improvements with negligible runtime overhead.

Area Analysis: gPIM introduces 32 PEs, a 4 KB buffer, and a scheduler to each vault of the logic layer. Based on Cadence [29] simulations, the area overhead of the gPIM logic layer for 32 vaults is 10.05 mm^2, which is under the 24 nm process technology. It only accounts for 1.03% of the total logic layer area.

Thermal Analysis: 3D stacking memory has strict thermal constraints, and the design of the logic layer could impact on thermal issues of HMC. Previous study shows that the power of the logic layer can not exceed 10W [32]. The power overhead of gPIM logic layer is 5.69 W, which is lower than the maximum power constraints.

VI. RELATED WORK

Systems and Accelerators for GCNs Inference: There are many software and hardware studies on GCNs. DGL [36] and PyG [23] are popular GCNs frameworks. They accelerate GCN workloads with hardware optimization functions, such as sparse matrix multiplication and scatter. NeuGraph [10] proposes a SAGA-NN programming model and supports multiple GPUs for large graphs on GCNs. PCGCN [12] partitions the graph into multiple dense graphs and sparse graphs according to the power-law distribution of natural world graphs, and processes them in batches to take advantage of the locality in graphs. GNNAdvisor [11] proposes an effective workload management runtime system to accelerate GCNs on the GPU. Prior studies reveal that special hardware architecture (such as ASIC and FPGA) design solutions can effectively improve performance [7], [8]. However, these earlier GCNs frameworks and special hardware designs still suffer from memory-bound inefficiencies. gPIM leverages a PIM-based architecture to resolve the memory challenge faced in the GCNs inference.

PIM-based Graph Processing: PIM is expected to address the memory bottleneck. Many PIM-based graph processing accelerators have been proposed. GraphR [37] is the first ReRAM-based accelerator for graph processing. Tesseract [26] proposes a programmable PIM parallel graph processing architecture. GraphP [17] further proposes to reduce the communication overhead between cubes by changing the data layout. GraphQ proposes a novel structured communication mechanism to eliminate irregular data movement between cubes. GraphPIM [38] offloads atomic operations on graph attribute data to HMC. Unlike graph processing, GCNs have hybrid characteristics that include not only irregular memory access but also intensive computations. Moreover, the vertex feature vector length is often long and available between layers. Therefore, previous PIM-based graph processing solutions are difficult to capture these unique characteristics raised in GCNs.

PIM-based Neural Network Acceleration: There are also many studies on PIM-based neural network accelerators. Liu et al. [39] propose a heterogeneous PIM architecture to accelerate neural networks. TETRIS [40] proposes a neural network accelerator using the 3D stack memory. ISAAC [41] proposes a ReRAM-based convolutional neural network accelerator. Unlike traditional neural networks, GCNs have an uncertain performance bottleneck combination phase and an irregular memory-bound aggregation phase. These PIM accelerators are hardly applied to accelerate the GCNs. In this paper, we use a GPU-PIM architecture to accelerate GCNs inference. The compute-intensive combination is deployed on GPU while memory-bound aggregation and combination are accelerated by PIM side.

VII. CONCLUSION

In this paper, we propose gPIM, which aims to accelerate GCNs inference on a GPU-PIM architecture. gPIM preserves GPU to perform compute-intensive combination while memory-bound aggregation and combination are offloaded to PIM side. To minimize the communication overheads between HMC cubes, a GCN-induced graph partitioning is developed. In addition, a programmer-transparent performance estimation mechanism is proposed to determine the performance bottleneck of combination and use the optimal engine to accelerate. Our results show that gPIM can outperform CPU, the state-of-the-art Nivida Tesla V100, and GCN accelerator AWB-GCN significantly in terms of both performance and energy savings.

REFERENCES

[1] H. Dai, Z. Kozareva, B. Dai, A. J. Smola, and L. Song, “Learning steady-states of iterative algorithms over graphs,” in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 1114–1122.
[2] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” 2016, *arXiv:1609.02907*.
[3] D. Duvenaud et al., “Convolutional networks on graphs for learning molecular fingerprints,” in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2015, pp. 2224–2232.
[4] “Graph nets library,” 2018. [Online]. Available: https://deepmind.com/research/open-source/graph-nets-library
[5] R. Zhu et al., “AliGraph: A comprehensive graph neural network platform,” *Proc. VLDB Endow*, vol. 12, no. 12, pp. 2094–2105, 2019.
[6] A. Lerer et al., “PyTorch-BigGraph: A large-scale graph embedding system,” 2019, *arXiv:1903.12287*.
[7] M. Yan et al., “HyGCN: A GCN accelerator with hybrid architecture,” in *Proc. Int. Symp. High Perform. Comput. Architecture*, 2020, pp. 15–29.
[8] T. Geng et al., “AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing,” in *Proc. IEEE/ACM 53rd Annu. Int. Symp. Microarchitecture*, 2020, pp. 922–936.
[9] D. Chen et al., “GraphFly: Efficient asynchronous streaming graphs processing via dependency-flow,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, pp. 632–645.

[10] L. Ma et al., "Neugraph: Parallel deep neural network computation on large graphs," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 443–458.
[11] Y. Wang et al., "GNNAdvisor: An efficient runtime system for GNN acceleration on GPUs," 2020, *arXiv:2006.06608*.
[12] C. Tian, L. Ma, Z. Yang, and Y. Dai, "PCGCN: Partition-centric processing for accelerating graph convolutional network," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2020, pp. 936–945.
[13] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore," *Commun. ACM*, vol. 52, pp. 65–76, 2009.
[14] D. Chen et al., "A general offloading approach for near-DRAM processing-in-memory architectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2022, pp. 246–257.
[15] J. D. Leidel and Y. Chen, "HMC-sim: A simulation framework for hybrid memory cube devices," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2014, pp. 1465–1474.
[16] K. Hsieh et al., "Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems," in *Proc. 43rd Annu. Int. Symp. Comput. Architecture*, 2016, pp. 204–216.
[17] M. Zhang et al., "Graphp: Reducing communication for PIM-based graph processing with efficient data partition," in *Proc. Int. Symp. High Perform. Comput. Architecture*, 2018, pp. 544–557.
[18] Y. Zhuo et al., "GraphQ: Scalable PIM-based graph processing," in *Proc. IEEE/ACM 52nd Annu. Int. Symp. Microarchitecture*, 2019, pp. 712–725.
[19] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2017, *arXiv:1706.02216*.
[20] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," *Proc. 7th Int. Conf. Learn. Representations*, 2019.
[21] X. Zhang, S. L. Song, C. Xie, J. Wang, W. Zhang, and X. Fu, "Enabling highly efficient capsule networks processing through a PIM-based architecture design," in *Proc. Int. Symp. High Perform. Comput. Architecture*, 2020, pp. 542–555.
[22] K. Kersting, N. M. Kriege, C. Morris, P. Mutzel, and M. Neuman, "Benchmark data sets for graph kernels," 2016. [Online]. Available: http://graphkernels.cs.tu-dortmund.de
[23] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric," 2019, *arXiv:1903.02428*.
[24] Micron Technology, Inc., "Hybrid memory cube specification 2.1," 2015. [Online]. Available: https://www.nuvation.com/sites/default/files/Nuvation-Engineering-Images/Articles/FPGAs-and-HMC/HMC-30G-VSR_HMCC_Specification.pdf
[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 17–30.
[26] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. 42nd Annu. Int. Symp. Comput. Architecture*, 2015, pp. 105–117.
[27] "Nvidia profiler," 2010. [Online]. Available: https://docs.nvidia.com/cuda/profiler-usersguide/index.html
[28] "Nvidia-smi," 2007. [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface
[29] "Gate-level simulation methodology - cadence," 2018. [Online]. Available: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/gate-level-simulation-wp.pdf
[30] P. Tsai, C. Chen, and D. Sánchez, "Adaptive scheduling for systems with asymmetric memory hierarchies," in *Proc. IEEE/ACM 51st Annu. Int. Symp. Microarchitecture*, 2018, pp. 641–654.
[31] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proc. Int. Conf. Parallel Architectures Compilation*, 2015, pp. 113–124.
[32] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proc. 23rd Int. Symp. High- Perform. Parallel Distrib. Comput.*, 2014, pp. 85–98.
[33] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with hybrid memory cubes," in *Proc. 22nd Int. Conf. Parallel Architectures Compilation Techn.*, 2013, pp. 145–155.
[34] PyTorch profiler, [Online]. Available: https://pytorch.org/docs/stable/autograd.html#profiler
[35] "Intel xeon e5–2680 v3," 2014. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/series/78583/intel-xeon-processor-e5-v3-family.html
[36] "Deep graph library," 2019. [Online]. Available: https://docs.dgl.ai
[37] L. Song, Y. Zhuo, X. Qian, H. H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *Proc. Int. Symp. High Perform. Comput. Architecture*, 2018, pp. 531–543.
[38] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks," in *Proc. Int. Symp. High Perform. Comput. Architecture*, 2017, pp. 457–468.
[39] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *Proc. IEEE/ACM 51st Annu. Int. Symp. Microarchitecture*, 2018, pp. 655–668.
[40] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and efficient neural network acceleration with 3D memory," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 751–764.
[41] A. Shafiee et al., "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. IEEE/ACM 43rd Annu. Int. Symp. Comput. Architecture*, 2016, pp. 14–26.

Hai Jin (Fellow, IEEE) received the PhD degree in computer engineering from the Huazhong University of Science and Technology, in 1994. He is a chair professor of computer science and engineering with the Huazhong University of Science and Technology (HUST), in China. In 1996, he was awarded the German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked with the University of Hong Kong between 1998 and 2000, and was a visiting scholar with the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is a fellow of CCF, and a life member of the ACM. He has co-authored more than 20 books and published more than 900 research papers. His research interests include computer architecture, parallel and distributed computing, Big Data processing, data storage, and system security.

Dan Chen received the BS degree from the North China Electric Power University, in 2018. He is currently working toward the PhD degree with the School of Computer Science and Technology, Huazhong University of Science and Technology, in China. His research interests include focus on processing-in-memory and graph neural network.

Long Zheng (Member, IEEE) received the PhD degree from the Huazhong University of Science and Technology (HUST), in 2016. He is currently an Associate Professor with the School of Computer Science and Technology, Huazhong University of Science and Technology. His current research interests include program analysis, runtime systems, and heterogeneous computing with a particular focus on graph processing.

Yu Huang received the PhD degree from the Huazhong University of Science and Technology (HUST), in 2022. He is now working toward the postdoctoral fellow with Zhejiang Lab, in China. His research interests include focus on processing-in-memory and graph processing.

Pengcheng Yao received the PhD degree from the Huazhong University of Science and Technology (HUST), in 2022. He is currently working toward the postdoctoral fellow with Zhejiang Lab, in China. His research interests include focus on graph processing and domain specific accelerator.

Jin Zhao received the PhD degree from the Huazhong University of Science and Technology (HUST), in 2022. He is currently working toward the postdoctoral fellow with Zhejiang Lab, in China. His current research interests include graph processing, system software, and architecture.

Xiaofei Liao (Member, IEEE) received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), China, in 2005. He is currently a professor with the School of Computer Science and Technology, HUST. His research interests include the areas of system virtualization, system software, and cloud computing.

Wenbin Jiang (Member, IEEE) received the PhD degree in information and communication engineering from the Huazhong University of Science and Technology (HUST), in 2004. He is currently a professor with the School of Computer Science and Technology, HUST. He was with Aizu University, Fukushima, Japan, for research visiting in 2005. He has published about 40 research papers. His current research interests include multimedia, ubiquitous computing, data management, and so on. He has been the PC Chair, the Publicity Chair, and a PC Member of more than 50 international conferences.