Exploiting Parallelism with Vertex-Clustering in Processing-In-Memory-based GCN Accelerators

Yu Zhu*, Zhenhua Zhu*, Guohao Dai, Kai Zhong, Huazhong Yang, Yu Wang
Department of Electronic Engineering, BNRist, Tsinghua University, Beijing, China
yu-wang@tsinghua.edu.cn

***Abstract*—Recently, Graph Convolutional Networks (GCNs) have shown powerful learning capabilities in graph processing tasks. Computing GCNs with conventional von Neumann architectures usually suffers from limited memory bandwidth due to the irregular memory access. Recent work has proposed Processing-In-Memory (PIM) architectures to overcome the bandwidth bottleneck in Convolutional Neural Networks (CNNs) by performing *in-situ* matrix-vector multiplication. However, the performance improvement and computation parallelism of existing CNN-oriented PIM architectures is hindered when performing GCNs because of the large scale and sparsity of graphs.**

To tackle these problems, this paper presents a *parallelism enhancement framework for PIM-based GCN architectures*. *At the software level*, we propose a fixed-point quantization method for GCNs, which reduces the PIM computation overhead with little accuracy loss. We also introduce the vertex clustering algorithm to the graph, minimizing the inter-cluster links and realizing cluster-level parallel computing on multi-core systems. *At the hardware level*, we design a Resistive Random Access Memory (RRAM) based multi-core PIM architecture for GCN, which supports the cluster-level parallelism. Besides, we propose a coarse-grained pipeline dataflow to cover the RRAM write costs and improve the GCN computation throughput. *At the software/hardware interface level*, we propose a PIM-aware GCN mapping strategy to achieve the optimal tradeoff between resource utilization and computation performance. We also propose edge dropping methods to reduce the inter-core communications with little accuracy loss. We evaluate our framework on typical datasets with multiple widely-used GCN models. Experimental results show that the proposed framework achieves $698\times$, $89\times$, and $41\times$ speedup with $7108\times$, $255\times$, and $31\times$ energy efficiency enhancement compared with CPUs, GPUs, and ASICs, respectively.

I. Introduction

Graph Neural Network (GNN) models have shown powerful learning capabilities in tasks on graphs, such as node classification and recommendation systems [1], [2]. Recently, GCNs have become the most popular branch of GNN models due to their lower computational overhead with outstanding accuracy [3]. The accuracy of GCNs is $10\% \sim 20\%$ higher than the traditional deep learning models on many graph datasets [4]. The typical GCN models consist of multiple layers, and each layer contains two steps: *Aggregation* and *Combination*, as shown in Fig. 1(a). In the former step, each vertex aggregates its neighbors' features, which can be expressed as sparse-matrix matrix multiplications (SpMMs) between the sparse adjacency matrix and the dense feature matrix. In the latter step, the aggregation results will be fed into a Multi-Layer-Perceptron (MLP) to update vertex features for the next layer.

In von Neumann architectures, GCN computing suffers from limited memory bandwidth because of the irregular memory access and massive data movements. Fig. 1(b) shows the relationship between the global load throughput and the number of node dimensions when computing GCNs with GPUs [5]. As can be seen, the global throughput of GCNs reaches to the global bandwidth bound when the feature dimensions grow larger, revealing the heavy burden of data movements.

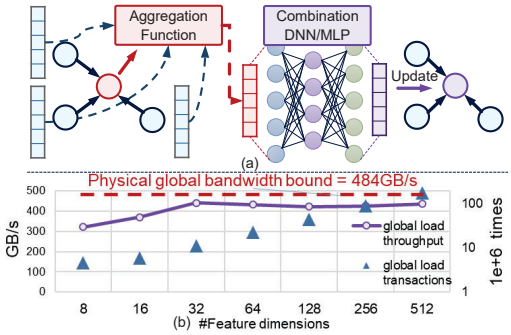


Fig. 1. (a) GCN computation flow; (b) Bandwidth limitation in GCNs [5].

Recently, Resistive Random Access Memory (RRAM) based Processing-In-Memory (PIM) architectures have shown great potential to reduce data movements by performing Matrix-Vector Multiplications (MVMs) in memory. Existing PIM-based CNN architectures can achieve tremendous speedup [6]. Inspired by the *in-situ* computing capability, PIM becomes one promising alternative to reduce the data movements and improve energy efficiency of GCNs.

However, directly adopting existing PIM architectures for GCNs faces two serious challenges. (1) The computation parallelism of PIM can not satisfy the computing requirements of the aggregation step. Existing PIM architectures use RRAM crossbars to store the matrix and perform *in-situ* MVMs. To deal with the aggregation step in GCNs, one SpMM needs to be split into massive MVMs that are executed sequentially in PIM, resulting in long processing time. For example, the SpMM of the aggregation step on a real application graph in ARXIV dataset [4] will be split into 169,343 sequential MVMs, bringing heavy computation burden on PIM. (2) The sparsity of the graph results in low hardware resource utilization [7], wasting the potential computation parallelism inside the memory crossbars. In existing PIM, the vector input of MVM is transferred into voltage vector to activate the corresponding crossbar rows. But when performing SpMM in PIM, the sparse input only activates very few crossbar rows (e.g., averagely 4.5 rows of 19,717 rows in PUBMED dataset [8]), leading to extremely low crossbar utilization.

To cope with these challenges, we propose a parallelism

*: Both authors contributed equally to this work.

enhancement framework for PIM-based GCN architectures. The main contributions include:

- At the algorithm level, we propose a GCN quantization method to transform the 32-bit floating-point graph data to the fixed-point form, which reduces the PIM hardware overhead with little accuracy loss ($< 1\%$). Further, we introduce the vertex-clustering algorithm to the graph, minimizing the inter-cluster links and realizing cluster-level parallel computing on multi-core systems.
- At the hardware level, we propose an RRAM based multi-core PIM architecture for GCNs (RP-GCN). RP-GCN utilizes the Network-on-Chip (NoC) structure to exploit the cluster-level computation parallelism. Furthermore, a coarse-grained pipeline dataflow is designed for RP-GCN to further improve the throughput.
- At the interface level, we propose PIM-aware GCN mapping strategy to explore the optimal tradeoff between resource utilization and performance. We also propose the edge dropping method to further reduce the inter-core communication costs with little accuracy loss ($< 1\%$).

II. Background and Related Work

A. Graph Convolutional Networks

GCNs have become the most popular GNN models due to their lower computational overhead with outstanding accuracy [3]. Original GCN model [1] contains multiple layers, and each of them can be expressed by the following formula:

$$H^{(l+1)} = \sigma(\tilde{A}H^{(l)}W^{(l)}). \tag{1}$$

In the equation, $H^{(l)}$, $W^{(l)}$, and σ represent the l^{th} layer's feature matrix, weight matrix, and activation function, respectively. $\tilde{A}$ is the normalized adjacency matrix which stays stable for a given graph and can be generated offline.

In another typical work GraphSAGE [9], researchers abstract the GCN model to a more general two-step form, which can be formulated as:

$$\begin{aligned} &Aggr.: a_u^{(k)} = AGGR(h_v^{(k)}, h_u^{(k)}), v \in \mathbb{N}(u) \\ &Comb.: h_u^{(k+1)} = COMB(a_u^{(k)}) \end{aligned} \tag{2}$$

in which $h_u^{(k)}$ represents the feature vector of vertex u in the k^{th} layer and $\mathbb{N}(u)$ is the set of the neighbors of vertex u. The computation flow is illustrated in Fig. 1(a). For each vertex, the aggregation function aggregates the feature vectors from its neighbors and send the output to the combination function to compute the feature vector of the vertex for the next layer. Additionally, GraphSAGE model only samples a few neighbors of each vertex to take part in the aggregation. Generally speaking, in most of the existing GCN models the function $AGGR$ serves as mean [1], sum [10], or max [9], which can be expressed as matrix operations between the weighted adjacency matrix $\tilde{A}$ and the feature matrix $H^{(k)}$ of the k^{th} layer. The $COMB$ function is mostly an MLP, transforming the aggregation results to new feature vectors.

B. RRAM and Processing-In-Memory

RRAM [11] is a kind of emerging non-volatile memory that uses conductance to store data. Benefit from the high storage density, GB-level RRAM storage is promising to become the next generation of non-volatile main memory [12].

In addition to the high storage density, RRAM and its crossbar structure provide alternative solutions to realize in-memory computing. In RRAM-based PIM architectures, the RRAM's conductance and word-line voltage represent the matrix and input vector, respectively, and the result of MVM is obtained from the bit-line current due to the Ohm's law. Recent work has proposed several RRAM-PIM-based CNN accelerators [6]. These accelerators reduce the heavy burden of data movement in CNN by performing *in-situ* MVMs, and achieve great improvement on speed and energy efficiency [6].

III. PIM-Aware GCN Algorithm Optimization

To realize PIM-based SpMM computation in GCN, we need to store one matrix in crossbars and convert the other one into serial voltage vectors to perform multiple *in-situ* MVMs. Because the vertex number is usually much larger than the feature dimension, (e.g., PUBMED has 19,717 vertices with the feature dimension of 500) and the graph structure is sparse (e.g., 99.97% elements are zero in the adjacency matrix of PUBMED), we store the feature matrix rather than the adjacency matrix in crossbars to reduce the storage burden and improve the resource utilization. To be specific, the dense feature matrix ($H_{N\times F}$, N: vertex number, F: feature dimension) is mapped onto crossbars, and the sparse adjacency matrix ($\tilde{A}_{N\times N}$) is split into input vectors and loaded vertex by vertex.

The graph data (i.e., feature and adjacency matrix) in GCNs are usually floating-point data (32-bit), while the precision of RRAM and input voltages are low (e.g., 2-bit). Therefore, using low-precision hardware for calculating high-precision GCNs suffers from heavy hardware overhead. To reduce the computing and storage burden, we use uniformed quantization on GCN and transform the floating-point graph data and weights into 8-bit fixed-point data. Detailed results displayed in Section VI-D show the proposed quantization decreases the storage overhead and improves the performance with little accuracy loss.

Furthermore, as shown in Fig. 2(a), due to the huge vertex number and the graph sparsity, existing PIM architecture takes enormous iterations to finish SpMM operations with extremely low resource utilization. To tackle these two problems, a natural idea is to partition the original graph into small and dense sub-graphs and process them in parallel on multi-core systems. However, there still exist edges connecting different sub-graphs. These edges cause the results of different sub-graphs need to be merged to generate the final results, resulting in additional communication costs among multiple cores. In order to fully utilize RRAM resources and improve computing parallelism of multi-core systems while minimizing the communication overhead, we model the graph partition problem as a vertex clustering problem. The optimization target of vertex clustering is to maximize the intra-cluster connections (i.e., edges inside each vertex cluster, indicate the resource utilization) with the minimum inter-cluster connections (i.e., edges among different vertex clusters, indicate the communication overhead). In this paper, we use *Graclus* [13], an efficient vertex clustering algorithm, to achieve the above clustering goal.

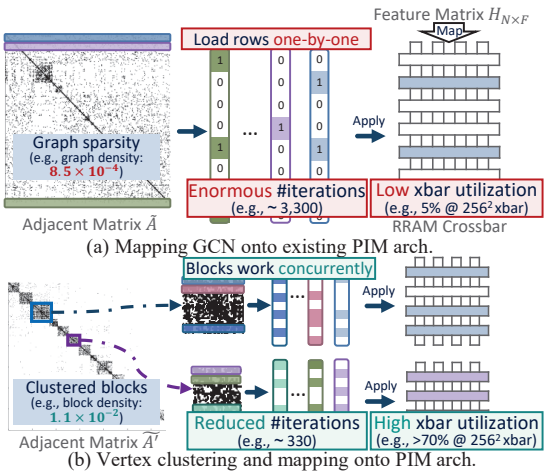


Fig. 2. Vertex-clustering-based mapping strategy. With the proposed strategy, the computation parallelism and the crossbar utilization are improved.

By introducing the clustering algorithm, we improve the computation parallelism and resource utilization for three reasons (shown in Fig. 2(b)): (1) most edges are allocated inside a vertex clustering, making the aggregation of multiple sub-graphs parallelizable; (2) the vertex number of each sub-graph is shrunk, reducing the iteration number in aggregation cores; (3) the denser sub-graphs improve the resource utilization of each core. Additionally, the computation complexity of *Graclus* is far lower than the GCN inference [13], making its overhead negligible. But there still exist few remaining edges among clusters after clustering, which will be handled in Section V.

IV. Hardware Design

A. Overall Architecture

At the hardware level, we design RP-GCN to realize highly parallel in-memory GCN computation, which is shown in Fig. 3(a). The architecture mainly includes four parts: a control block, an aggregation core array, a combination core array, and two buffers (i.e., one neighbor buffer for adjacency vector storage and one intermediate buffer). The control block consists of an instruction queue and a decoder.

B. Aggregation Core Array

The aggregation core array consists of multiple RRAM-based aggregation cores connected in an NoC pattern.

In the aggregation step, the feature matrix is stored in RRAM crossbars statically, while the vectors of the adjacency matrix are dynamically loaded from the neighbor buffer to aggregation cores. In each iteration, multiple vectors are loaded and transmitted to different cores by the data forwarding units in NoC. To exploit the cluster-level computation parallelism brought by the clustering algorithm, different cores can work concurrently and independently. Since the vertices may need multiple cores for aggregation because of the inter-cluster edges, data forwarding units in NoC have the capability of merging cores' results, which can also work in parallel. Finally, the merged aggregation results are sent to the intermediate buffer for the following MLP operations.

At the end of each GCN layer, the updated feature matrix will be derived from the combination step and needs to be written back to the aggregation core array. For the purpose of distributing the intensive feature updates at the end of each layer into multiple iterations, we divide the aggregation core into two parts, i.e., the work space and the update space, which work in a ping-pong fashion. When the work space performs vertex aggregations, the update space can update the feature vectors of previous vertices simultaneously, which have completed the current layer's computation.

The details of aggregation cores are shown in Fig. 3(a-A). Each core contains several RRAM crossbars with peripheral circuits, i.e., Digital-to-Analog Converters (DACs), Analog-to-Digital Converters (ADCs), and multiplexers (MUXes). To support different aggregation functions, comparators and adders are placed near RRAM crossbars, and results from multiple crossbars will be merged iteratively with comparators or adders.

The design of data forwarding units is illustrated in Fig. 3(a-B). The data forwarding units are responsible for receiving data from different source cores, merging the intermediate data, and transferring the results to different destination cores.

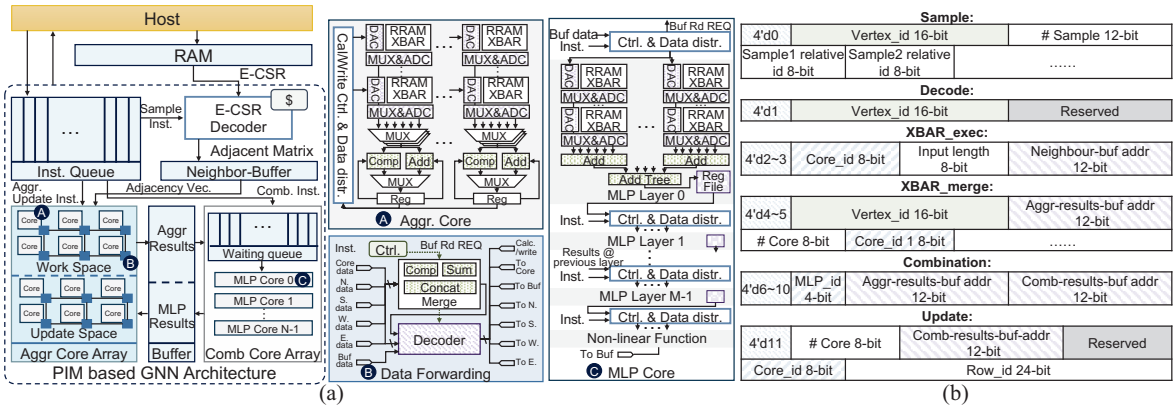
C. Combination Core Array

As shown in Fig. 3(a-C), the combination core array contains a waiting queue and many MLP cores, computing combination steps in parallel to succeed the cluster-level parallelism from the aggregation core array. The MLP core respectively uses RRAM crossbars and look-up-tables for MLP computations and non-linear functions.

In the combination step, the inputs of combination, i.e., the aggregation results, are firstly read from the intermediate buffer into the waiting queue. For each GCN layer, each MLP core stores the entire MLP weights of it. Therefore, the results popped from the waiting queue are sent to any idle MLP core in this layer. Besides the parallelism of multi-core, the throughput is further improved by the pipelined layer execution manner within each MLP core. Finally, the results of non-linear functions are delivered to the intermediate buffer for updating feature matrices in the aggregation core array.

D. Dataflow and Instruction Set Design

Benefit from the ping-pong fashion of the aggregation core array, aggregation, combination, and feature update of different iterations can be executed concurrently. To increase the throughput of GCN computation, we design a coarse-grained intra-layer pipeline dataflow. In this pipeline dataflow, different GCN layers work sequentially, while the operations inside each layer work in the pipeline manner. In the pipeline structure, when the work space of the aggregation core array is performing the $(k+2)^{th}$ aggregation, the combination core array is performing the $(k+1)^{th}$ combination and the update space of the aggregation core array is performing the k^{th} update. The three steps of different iterations work concurrently, and each iteration process a part of the whole graph. Therefore, the distributed update overhead can be hidden by aggregation and combination, improving the entire throughput of RP-GCN. Simulation results show that the aggregation step takes a little longer time than the combination step in each iteration, and



(a) (b)

Fig. 3. (a) RP-GCN architecture, with: **A.** RRAM based PIM cores for aggregation, **B.** data forwarding modules for core communication, and **C.** PIM cores for MLP; (b) Instruction set design.

the latency of feature update depends on the dimension of the feature to be updated. In datasets with smaller hidden channel size (e.g., PUBMED), the aggregation step is most time-consuming, while in datasets with larger feature size (e.g., ARXIV), the update step becomes the bottleneck.

We also propose an instruction set to control the computation flow of RP-GCN, as illustrated in Fig. 3(b). In the compilation stage, the GCN model is translated into instruction packages by the host. These instructions are buffered in the instruction queue, then fetched by the control block and sent to aggregation core array and the combination core array.

V. Mapping Strategy and Optimization

Despite the virtues of the clustering method described in Section III, the unawareness of the PIM architecture during vertex clustering brings two new issues. Firstly, the cluster size varies greatly after clustering (e.g., the cluster size ranges from 14 to 1,403 after clustering the graph from PUBMED dataset to 128 clusters), while the aggregation core size is fixed, causing the waste and unbalanced usage of computing resources. Secondly, the remaining edges among different clusters cause inter-core communication overhead. To solve these issues, we present the PIM-aware GCN mapping strategy and edge dropping method, as shown in Algorithm 1.

A. PIM-Aware GCN Mapping Strategy

The mapping strategy aims to map the vertex feature onto crossbar rows with high resource utilization and low computing/communication latency, which contains two steps.

Firstly, the mapping strategy solves the contradiction between the various cluster size and the fixed row number R in aggregation cores. We split the feature matrix of large clusters into multiple cores (**line 4**∼**line 7**) and merge multiple feature matrices of small clusters into one core (**line 8**∼**line 11**). Before mapping we sort the clusters according to the cluster size to make the merge results more compact (**line 2**). This step improves the crossbar utilization with a little parallelism degradation costs (i.e., multiple small clusters are stored in one aggregation core which can not be computed in parallel).

Secondly, the mapping strategy use the unused rows to reduce the inter-cluster communication costs. After the clustering, there still exist external edges among different clusters. The inter-cluster edges mean that the features of one vertex's neighbors are distributed in multiple cores, leading to the inter-core communications in the aggregation step of this vertex. We notice that many aggregation cores contain idle rows when the number of stored vertices cannot exactly equal the row number. This hardware idleness inspires us to reduce the inter-core communication by duplicating and mapping the features of neighbors outside the cluster into the idle part of the core (**line 15**∼**line 21**). This step can improve the crossbar utilization while reducing the communication overhead.

After taking these two mapping steps, we make full use of the hardware resource and reduce the aggregation latency. For example, the mapping strategy can achieve 60% crossbar utilization improvement and $1.56\times$ speedup on CORA.

B. Edge Dropping Method

Another way to reduce the communication cost is to reduce the number of inter-cluster edges. Recently research has suggested that randomly dropping some edges of the graph is benefit to the accuracy of GCN [14]. Based on this idea, we propose the edge dropping strategy by delete some of the inter-core edges to reduce the computation dependency. As shown in Algorithm 1 **line 23**∼**line 30**, we use a set S to represent the links to be dropped, together with a threshold T to limit the dropping rate of each vertex. In small datasets (e.g., CORA, CITESEER and PUBMED), dropping all the inter-core edges will only result in an accuracy loss of less than 1%, while in other datasets such as ARXIV, the dropping rate threshold should be carefully chosen to ensure the accuracy. Detailed experiments will be shown in Section VI-D. By applying the edge dropping strategy on CITESEER dataset, we remove all the inter-core edges and the latency is decreased by 30%, with 0.8% accuracy loss.

VI. Evaluations

A. Experiment Setup

We use the GCN [1] model on five datasets, i.e., CORA [15], CITESEER [16], PUBMED [8], ARXIV [4], and REDDIT [17] as the evaluation benchmarks. The dataset parameters are shown in Table I. The data of RRAM, ADCs, and DACs refer to [18]–[20], respectively. The digital parts are synthesized by Synopsys Design Compiler® at TSMC 65nm technology node with 300MHz. We use CACTI [21] for the buffer simulation with the buffer size of 64KB. We evaluate GCNs with DGL

Algorithm 1 Mapping Strategy and Optimization

Input: $Cluster_i$: Vertex sets of clusters; R: The number of core rows; T: Edge dropping threshold;
Output: $Core_i$: Vertex sets of cores;
/***1.PIM-Architecture-Aware Mapping Strategy:***/
Sort($\{Cluster_i\}$); $p = 1$; $i = 1$;
while $i \leq N_{cluster}$ **do**
  if $Size(Cluster_i) > R$ **then**
    $l_i = \lceil Size(Cluster_i)/R \rceil$;
    Map($Cluster_i$, $Core_{p \sim p+l_i}$);
    $p = p + l_i$;
  else
    $j = FindMax(i, R)$;
    Map($Cluster_{i \sim j}$, $Core_p$);
    $p = p + 1$; $i = j$;
  end if
  $i = i + 1$;
end while
for i = 1 to N_{core} **do**
  $rest_i = R - Size(Core_i)$; $S' = S = \{\}$;
  while $Size(S') < rest_i$ **do**
    $S = S'$; $S' = MinIncrease(Core_i, S)$;
  end while
  $Core_i = Core_i \cup S$;
end for
/***2.Edge Dropping Strategy***/
for i = 1 to N_{vertex} **do**
  $S = \mathbb{N}(i)$;
  while $Size(S) >= T * Size(\mathbb{N}(i))$ **do**
    $Cid = MaxIntersection(S, Core)$;
    $S = S - Core_{Cid}$;
    $WL_{Cid} = WL_{Cid} \cup i$;
  end while
end for

TABLE I
BASIC SETTINGS OF THE DATASETS.

	CORA	CITESEER	PUBMED	ARXIV	REDDIT
Vertices	2708	3327	19717	169343	232965
Edges	10556	9228	88651	1166243	114848857

[22] on both 40-core Intel Xeon E5-2630 v4 2.20GHz CPU and Nvidia RTX 2080Ti GPU. We also compare RP-GCN with PRIME [6], which maps the features to the cores in order and uses the same hardware configurations as RP-GCN.

B. Cross-Platform Comparison

We first compare the performance of our architecture with the CPU, GPU, and PRIME on four datasets. As illustrated in Fig. 4(a), PRIME only has $44 \sim 86\times$ speedup compared to the CPU when processing GCNs, which is much lower than processing CNNs ($1596 \sim 11802\times$ in [6]), due to the low resource utilization brought by the graph sparsity as described in Section III. By applying our designs and optimizations, RP-GCN achieves $87 \sim 664\times$ speedup on the four datasets compared with CPU implementation, as well as $45 \sim 120\times$ speedup compared with GPU implementation. Among the four datasets, RP-GCN achieves the best speedup ratio on ARXIV, which reveals our ability on processing large-scale graphs. As for the energy results displayed in Fig. 4(b), the energy efficiency of RP-GCN is raised by $2834 \sim 7108\times$ and $172 \sim 606\times$ compared with CPUs and GPUs, respectively. The PIM-based architectures reduce a large amount of data movement overhead, leading to the superior energy efficiency compared with CPUs and GPUs. Additionally, Table II show the speedup and energy results of RP-GCN compared with

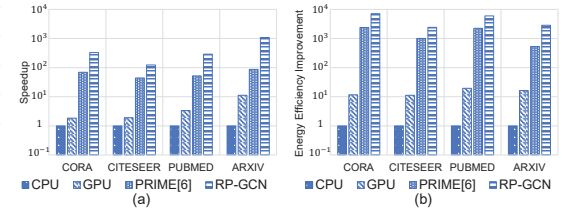


Fig. 4. (a) Speedup and (b) energy efficiency results normalized to CPU.

TABLE II
SPEEDUP AND ENERGY EFFICIENCY RESULTS ON REDDIT COMPARED TO EXISTING GCN ACCELERATORS NORMALIZED TO CPU.

	HyGCN [23]	AWB-GCN [24]	**RP-GCN**
Speedup	2.5×	14.7×	**608.1**×
Energy Efficiency	44.9×	12.6×	**1405.8**×

existing GCN accelerators. The proposed architecture achieves $205\times$ and $41\times$ speedup with $31\times$ and $112\times$ energy efficiency increase compared with HyGCN [23] and AWB-GCN [24] on REDDIT dataset, respectively.

C. Performance Ablation

In Section V, we take two optimization strategies to improve the performance of RP-GCN, i.e., the mapping strategy and the edge dropping method. In order to clarify the performance gain of each strategy, we propose two different performance ablation settings with an ordinal-mapping baseline (i.e., Prime) and evaluate their performance on four datasets, as shown in Table III. The proposed mapping and edge dropping both bring large performance gains compared to the baseline, they increase the inference speed by $4.22\times$ and $1.91\times$ averagely.

D. Influences on Accuracy

We mainly consider two factors that influence the GCN accuracy, i.e., the edge dropping strategy and the quantization error. As described in Section V, we remove some of the inter-core edges and set a threshold T to limit the dropping rate and control the accuracy loss. So there exists a tradeoff between the latency and accuracy: a higher dropping rate threshold T usually means a lower latency with a lower accuracy. As for the quantization error, we analyze two error sources. From the algorithm perspective, the high-precision floating-point input vectors and weight matrices are transformed to lower-precision fixed-point data, leading to the quantization error. From the hardware perspective, due to the limited ADC resolution and other hardware restrictions, the quantization error accumulates during the MVMs.

The evaluation results are shown in Fig. 5. In all the datasets, the latency decreases when T increases. However, when T grows larger, the accuracy also starts to decrease. On the one hand, in CORA, CITESEER, and PUBMED, setting T to 1 will not reduce the accuracy by more than 1%, and by removing all of the inter-core links, the latency is reduced by 6%, 30%, and 75%, respectively. On the other hand, in ARXIV and REDDIT, dropping too many edges will lead to a relatively severe accuracy loss. To limit the accuracy loss below 1%, the threshold T should not be higher than 0.2 for ARXIV or 0.5 for REDDIT, as represented by the gray lines. Generally speaking, higher data precision and ADC resolution are beneficial to the

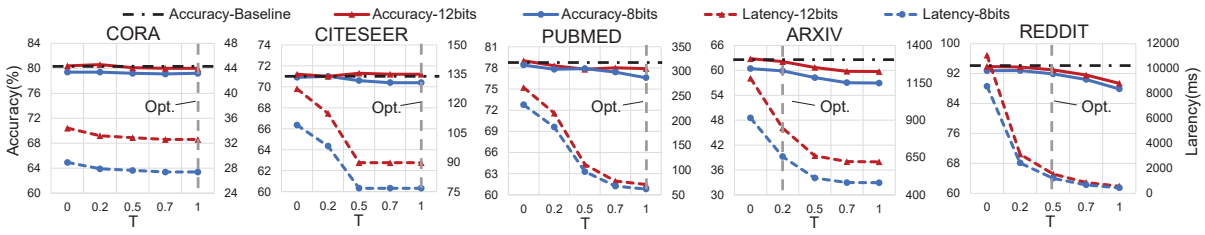


Fig. 5. The trade off between latency and accuracy. T represents the edge dropping rate threshold. More edges will be dropped when T grows larger. Each gray line represents the optimal T that makes the accuracy loss less than 1% and minimizes the latency.

TABLE III
PERFORMANCE GAINS BROUGHT BY THE OPTIMIZATIONS, COMPARED WITH ORDINAL-MAPPING BASELINES. M. REPRESENTS MAPPING STRATEGY AND D. REPRESENTS EDGE DROPPING.

	CORA	CITESEER	PUBMED	ARXIV
w/ M., w/o D.	4.47	1.96	1.48	8.97
w/ M. and D.	4.72	2.79	5.59	12.49

accuracy. As demonstrated in Fig. 5,in CORA and CITESEER, using 8-bit data is enough to keep the accuracy loss less than 1%. However, PUBMED, ARXIV, and REDDIT need at least 12-bit data to achieve acceptable accuracy.

E. Overhead Breakdown

We analyze the energy and area consumption of our design on PUBMED dataset. We breakdown the total energy and area into five parts, the buffer, the RRAM crossbars, the ADCs, the DACs, and the others (e.g., the NoCs , the adders, etc.). Among them, the ADCs consume the most energy of nearly 50%. Besides, the buffer also consumes 26% energy. As for the area results, the ADCs and DACs takes 58% and 28% of the area, respectively, while the RRAM crossbars consume 13%.

VII. CONCLUSION

In this work, we propose a parallelism enhancement framework for PIM-based GCN architectures. At the software level, we propose a quantization method to reduce the computation overhead and introduce a vertex clustering algorithm to realize cluster-level parallel computing on multi-core systems. At the hardware level, we propose RP-GCN and a pipeline dataflow to leverage the computation parallelism for GCNs. At the interface level, we design the PIM-architecture-aware mapping strategy and the edge dropping method to achieve the optimal tradeoff between resource utilization and computation performance. Experiments show that the proposed framework achieves $698\times$, $89\times$, and $41\times$ speedup with $7108\times$, $255\times$, and $31\times$ energy efficiency enhancement compared with CPUs, GPUs, and ASICs, respectively. Furthermore, the proposed GCN algorithm optimization and vertex-clustering based mapping strategy can also be applied to other GCN accelerator designs, not limited to PIM architectures.

VIII. ACKNOWLEDGEMENTS

This work was supported by National Natural Science Foundation of China (No. 61832007, U19B2019, 62104128), National Key RD Program of China (No. 2017YFA02077600); China Postdoctoral Science Foundation (No. 2019M660641); Tsinghua EE Xilinx AI Research Fund; Beijing National Research Center for Information Science and Technology (BNRist); Beijing Innovation Center for Future Chips; T-Head Semiconductor Co., Ltd. of Alibaba Group.

REFERENCES

[1] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.
[2] R. Ying, R. He, K. Chen, *et al.*, "Graph convolutional neural networks for web-scale recommender systems," in *ACM KDD*, pp. 974–983, 2018.
[3] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, "Computing graph neural networks: A survey from algorithms to accelerators," *arXiv preprint arXiv:2010.00130*, 2020.
[4] W. Hu, M. Fey, M. Zitnik, *et al.*, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.
[5] G. Huang, G. Dai, Y. Wang, and H. Yang, "Ge-spmm: general-purpose sparse matrix-matrix multiplication on gpus for graph neural networks," *SC*, pp. 1–12, 2020.
[6] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ISCA*, pp. 27–39, 2016.
[7] G. Dai, T. Huang, Y. Wang, H. Yang, and J. Wawrzynek, "Graphsar: a sparsity-aware processing-in-memory architecture for large-scale graph processing on rerams," in *ASPDAC*, pp. 120–126, 2019.
[8] P. Sen, G. Namata, M. Bilgic, *et al.*, "Collective classification in network data," *AI magazine*, vol. 29, no. 3, pp. 93–93, 2008.
[9] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NeurIPS*, pp. 1025–1035, 2017.
[10] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," *ICLR*, 2018.
[11] H.-S. P. Wong, H.-Y. Lee, S. Yu, *et al.*, "Metal–oxide rram," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.
[12] T.-y. Liu, T. H. Yan, R. Scheuerlein, *et al.*, "A 130.7 mm2 2-layer 32 gb reram memory device in 24 nm technology," *IEEE JSSC*, vol. 49, no. 1, pp. 140–153, 2013.
[13] I. S. Dhillon, Y. Guan, and B. Kulis, "Weighted graph cuts without eigenvectors a multilevel approach," *IEEE TPAMI*, vol. 29, no. 11, pp. 1944–1957, 2007.
[14] Y. Rong, W. Huang, T. Xu, and J. Huang, "Dropedge: Towards deep graph convolutional networks on node classification," in *ICLR*, 2019.
[15] A. McCallum, "Cora dataset," 2017.
[16] C. L. Giles, K. D. Bollacker, and S. Lawrence, "Citeseer: An automatic citation indexing system," in *Proceedings of the third ACM conference on Digital libraries*, pp. 89–98, 1998.
[17] P. Yanardag and S. Vishwanathan, "Deep graph kernels," in *ACM KDD*, pp. 1365–1374, 2015.
[18] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE TCAD*, vol. 31, no. 7, pp. 994–1007, 2012.
[19] A. T. Ramkaj, M. Strackx, M. S. Steyaert, and F. Tavernier, "A 1.25-gs/s 7-b sar adc with 36.4-db sndr at 5 ghz using switch-bootstrapping, uspc dac and triple-tail comparator in 28-nm cmos," *IEEE JSSC*, vol. 53, no. 7, pp. 1889–1901, 2018.
[20] S. Mahdavi, R. Ebrahimi, A. Daneshdoust, and A. Ebrahimi, "A 12bit 800ms/s and 1.37 mw digital to analog converter (dac) based on novel rc technique," in *IEEE ICPCSI*, pp. 163–166, 2017.
[21] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *IEEE/ACM MICRO*, pp. 3–14, 2007.
[22] M. Wang, D. Zheng, Z. Ye, *et al.*, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.
[23] M. Yan, L. Deng, X. Hu, *et al.*, "Hygcn: A gcn accelerator with hybrid architecture," in *IEEE HPCA*, pp. 15–29, 2020.
[24] T. Geng, A. Li, R. Shi, *et al.*, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *IEEE/ACM MICRO*, pp. 922–936, 2020.