IMGA: Efficient In-Memory Graph Convolution Network Aggregation With Data Flow Optimizations

Yuntao Wei, Xueyan Wang, *Member, IEEE*, Shangtong Zhang, Jianlei Yang, *Senior Member, IEEE*, Xiaotao Jia, *Member, IEEE*, Zhaohao Wang, *Senior Member, IEEE*, Gang Qu, *Fellow, IEEE*, and Weisheng Zhao, *Fellow, IEEE*

***Abstract*—Aggregating features from neighbor vertices is a fundamental operation in graph convolution network (GCN). However, the sparsity in graph data creates poor spatial and temporal locality, causing dynamic and irregular memory access patterns and limiting the performance of aggregation on the Von Neumann architecture. The emerging processing-in-memory (PIM) architecture is based on emerging nonvolatile memory (NVM), like spin-orbit torque magnetic RAM (SOT-MRAM), and demonstrates promising prospects in alleviating the Von Neumann bottleneck. However, the limited memory capacity of PIM medium still incurs non-negligible data movements between PIM architecture and external memory. To solve this challenge, we propose an SOT-MRAM-based in-memory computing architecture, called IMGA, for efficient in-situ graph aggregation. Specifically, we design adaptive data flow management strategies that reuse vertex data in MRAM when processing graphs of different scales and adopt edge data as the control signal source to utilize the graph's structural information. A reordering optimization strategy leveraging hardware–software co-design principle is proposed to further reduce the costly data movement. Experimental results demonstrate that IMGA achieves an average 2523× and 21× speedup, and 1.03E+6 and 1.04E+3 energy efficiency compared with CPU and GPU, respectively.**

***Index Terms*—Data flow, graph aggregation, processing in memory (PIM), spin-orbit torque magnetic RAM (SOT-MRAM).**

I. Introduction

GRAPH is a fundamental data structure that can model real-world relationships, with broad applications in recommendation systems, biology, fraud detection, etc. Graph convolution network (GCN) has been proven to have great promise in feature extraction from graph data [1]. GCN mainly consists of two phases: 1) *Combination* and 2) *Aggregation* [2]. The *Combination* phase is similar to multilayer perceptron operation in the conventional neural network (NN). Every destination vertex gathers features from its source vertices in the *Aggregation* phase. The *Aggregation* stage enables the network to learn the topological connections of graphs, which is the main distinction between GCN and NN. However, the graph data with its sparsity shows poor spatial and temporal locality during *Aggregation* phase in the traditional Von-Neumann architecture, which further induces frequent random memory access and cache miss. Therefore, inefficient memory bandwidth usage [3] and insufficient data reuse in *Aggregation* phase lead to significant idle time in execution units when waiting for data available. Thus, the *Aggregation* phase becomes the performance bottleneck of GCN. Owing to the ever-increasing importance and the demand for efficient execution, hardware acceleration for GCN has drawn tremendous attention in recent years [2], [4], [5], [6], [7]. Traditional acceleration strategies, based on Von-Neumann architecture, usually implement customized execution pipelines and memory organizations to reduce the irregular accesses and synchronization overheads. However, the problems caused by poor spatial locality and temporal locality are difficult to be solved thoroughly.

The processing-in-memory (PIM) accelerators [8], [9], [10], [11], [12], [13], [14] have demonstrated good promise by realizing logic operation and arithmetic computation closer within the memory to leverage the sizeable internal bandwidth and inherent parallelism of the memory systems. Several PIM-based accelerators have been proposed in the past few years for efficient GCN computing [15], [16], [17], [18], [19], and they have achieved considerable performance compared with the accelerators based on Von-Neumann architecture.

However, the limited memory resources in these PIM accelerators make it impractical to store the whole large graph data with millions of vertices. Therefore, some works, such as

Manuscript received 4 January 2023; revised 15 April 2023; accepted 28 May 2023. Date of publication 21 June 2023; date of current version 22 November 2023. The work of Xueyan Wang was supported by the National Natural Science Foundation of China under Grant 62004011. The work of Jianlei Yang was supported by the National Natural Science Foundation of China under Grant 62072019. The work of Xiaotao Jia was supported in part by the National Natural Science Foundation of China under Grant 62006011, and in part by the Joint Funds of the National Natural Science Foundation of China under Grant U20A20204. This article was recommended by Associate Editor A. Gamatie. *(Corresponding authors: Xueyan Wang; Weisheng Zhao.)*

Yuntao Wei, Xueyan Wang, Shangtong Zhang, Xiaotao Jia, Zhaohao Wang, and Weisheng Zhao are with MIIT Key Laboratory of Spintronics, School of Integrated Circuit Science and Engineering, Beihang University, Beijing 100191, China (e-mail: wangxueyan@buaa.edu.cn; weisheng.zhao@buaa.edu.cn).

Jianlei Yang is with the School of Computer Science and Engineering, BDBC, State Key Laboratory of Software Development Environment, Beihang University, Beijing 100191, China.

Gang Qu is with the Department of Electrical and Computer Engineering and Institute for Systems Research, University of Maryland at College Park, College Park, MD 20742 USA.

Digital Object Identifier 10.1109/TCAD.2023.3288509

TARe [19], perform *Aggregation* by storing edge data in the PIM memory and using vertex data stored in external memory as input of PIM accelerator. However, edge data in the PIM accelerator have to be replaced frequently when traversal of large graph is needed in GCN computing. Under this circumstance, the edge data are seldom reused. Besides, the vertex data in GCN is represented as high-dimension vectors, which induce huge data movement overhead when read from external memory.

Recently, some architectures, such as PIM-GCN [18], stores the vertices in the PIM memory. Although these architectures achieve reuse of vertices data, there are still several challenges to further reduce data movement overhead. First, the reuse of vertices data depends on a significant proportion of vertices stored in PIM memory which is difficult for some large graphs. The PIM architecture should void unnecessary storage to reserve space for more vertices data. Second, when it is impossible to store all vertices in PIM memory, access to external memory is inevitable, which is highly cost in both time and energy. The data flow of the PIM accelerator should be designed to decrease these access. Third, the complex vertices data reuse strategy needs a more delicate control mechanism, which brings significant extra overhead in PIM architecture.

We propose a spin-orbit torque magnetic RAM (SOT-MRAM)-based PIM *Aggregation* acceleration architecture called IMGA to overcome these challenges.

To deal with the first challenge, the edge data is stored in external memory (such as DRAM) rather than MRAM in our architecture. The edge data is accessed sequentially during *Aggregation*, therefore the bandwidth of external memory can be utilized effectively. Besides, we propose an adaptive data flow management strategy with reserved space (RS) design to avoid unnecessary vertices data storage. To address the second challenge, we propose a reordering optimization algorithm to reduce vertices data access in external memory. The third challenge related to the control mechanism is tackled by treating edge data as control instructions rather than operands. We use nonvolatile SOT-MRAM for its fast write speed, low-energy cost, and high-write endurance. Thus, SOT-MRAM is promising for being the memory device of PIM architecture for large-scale graph aggregation.

We convert edge data to control signals with decode circuit and the complex instruction mechanism is avoided.

The contributions of this article can be summarized as follows.

1) We analyze the performance bottleneck of PIM-based architecture when processing different scale graphs and design adaptive data flow to reduce data movement overhead.
2) We propose a PIM-based *Aggregation* acceleration architecture called IMGA which is integrated with adaptive data flow. IMGA uses edge data as the control signal to better utilize graph structure information and to design a more simplified control mechanism.
3) We propose reordering optimization based on the power-law degree distribution of graphs to improve the performance of IMGA from the aspect of hardware–software co-design.

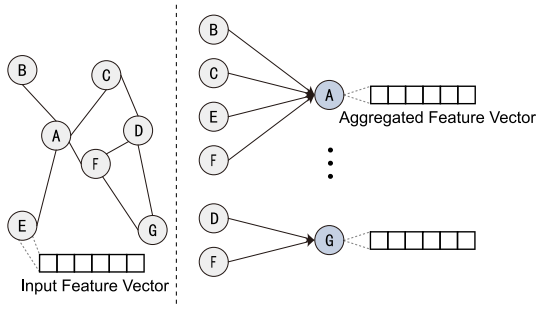

Fig. 1. Illustrative example of *Aggregation* phase of GCN.

The remainder of this article is organized as follows. Section II introduces the preliminaries of this article regarding GCN and SOT-MRAM. Section III elaborates the proposed in-memory *Aggregation* acceleration architecture. Section IV demonstrates the experimental results. Section V further discusses about some insights in adopting the proposed engine in GCN tasks and Section VI concludes this article.

II. Preliminaries

A. GCN and Aggregation

In GCN, as shown in Fig. 1, each vertex is represented as a feature vector with high dimensions. Every vertex needs to aggregate neighbors' feature vectors in the *Aggregation* phase to learn the topology structure of the graph and multiplex with weight matrix in the *Combination* phase. A simple *Aggregation* function can be defined as the summation of feature vectors, which can be computed as a matrix multiplication between the adjacency matrix and the feature matrix that consists of feature vectors of all the vertices. The equations of *Aggregation* and GCN computing can be rewritten as

$$X_a^l = AX^l \tag{1}$$

$$X^{l+1} = \sigma\left(X_a^l W\right). \tag{2}$$

In (1) and (2), X^l denotes input feature matrix, while X_a^l is the output feature matrix after *Aggregation* phase. W is weight matrix and $\sigma(\cdot)$ is activation function. There are several problems caused by this *Aggregation* equation. The adjacency matrix A used in *Aggregation* is highly sparse. Most elements in A are zero, meaning there is a large amount of useless computation. However, the wildly used sparse matrix compression format, such as compressed sparse column (CSC) and compressed sparse row (CSR), are not supported in this way of computing. Besides, the storage and transportation overhead of A is also significant. Even though the adjacency matrix can be split in many shards [11] and the shard without edge can be skipped during computing, the compression rate of this format is still lower than CSC or CSR. As shown in Table I, The first and second columns are the edge data compression rates of CSR format and shard with the size of 8×8, respectively. The shards format has a lower-compression rate than CSR in all the datasets. The third column in Table I is the ratio of shards with only one edge to the total number of shards, which means most shards are still sparse.

TABLE I
COMPARISON OF COMPRESSION RATE OF EDGE DATA IN CSC FORMAT AND 8 × 8 SHARDS

Datasets	CSC/%	8x8 Shards/%	One-edge/%
Cora(CR)	5.79	13.54	81.46
Pubmed(PB)	0.89	2.7	96.69
Citeseer(CS)	3.59	8.71	89.74
Ogbn-arxiv(OA)	0.15	0.49	96.94
Ogbl-collab(OC)	0.15	0.44	85.58
Ogbl-ppa(OP)	0.42	1.6	99.01
Reddit(RD)	6.77	22.95	84.41

The other way to perform *Aggregation* is vector addition under the control of the adjacency matrix. The *Aggregation* of *i*th vertex can be written as

$$x_{ia}^{l} = \sum_{j \in N(i)} x_{j}^{l}. \tag{3}$$

$N(i)$ denotes the set of vertex i's neighbors. In the (3), the adjacency matrix is converted as control signals rather than operand. By doing this, we can avoid unnecessary computing and compress the adjacency matrix in CSC or CSR format. However, due to the graph's sparsity, vectors that participate in an *Aggregation* operation are often not stored in memory continuously. The bandwidth of external memory is not fully utilized under this circumstance. More seriously, the high-cache-miss rate caused by the poor temporal locality of the graph data leads to frequent access to external memory. Thus, improving vertices data reuse is the key to high-performance computing of *Aggregation* in this way.

B. In-Memory Computing and SOT-MRAM

In traditional Von-Neumann architecture, data needs to be transferred from the main memory to the processing unit, which incurs long memory access latency and energy. The characteristic of irregular and random memory access patterns of graph computing makes this situation even worse. For large-scale graph computing, traditional architecture faces challenges in achieving high bandwidth. In the last two decades, in-memory computing has become a viable way to solve the memory and power wall, for it realizes logic units within memory. In this way, it can take advantage of inherent parallelism and large internal memory bandwidth, saving significant off-chip data communication energy and delay.

SOT-MRAM technology is a promising candidate for both last-level cache and main memory [20], [21], [22], [23]. A typical SOT-MRAM bit-cell is a composite structure of spin hall metal (SHM) and magnetic tunnel junction (MTJ). The relative parallel magnetization in both magnetic layers can be stable in parallel (P state) or anti-parallel (AP state), corresponding to a low-resistance (R_P) or a high-resistance (R_{AP}), respectively. Each SOT-MRAM cell located in computational subarrays is controlled by the write word line (WWL), read word line (RWL), write bit line (WBL), read bit line (RBL), and source line (SL). In summary, SOT-MRAM has the benefits of nonvolatility, low-switching energy, high endurance and retention time, high integration, and compatibility with CMOS technology. As a matter of fact, MRAM technology is undergoing the process of commercialization [24].

III. IMGA ARCHITECTURE

In this section, we first introduce the adaptive data flow designed to improve the vertex data reuse in graphs with different scale. Then, we design a custom edge data format to support our control mechanism. After that, we illustrate the overview of our IMGA architecture based on the adaptive data flow and the control mechanism. Finally, we propose a reordering optimization technique for IMGA architecture to improve the vertex data reuse from the aspect of hardware–software co-design.

A. Adaptive Data Flow

As each vertex can be accessed several times during the *Aggregation* phase, we decide to keep vertices data in IMGA's MRAM to reduce expensive data movement between MRAM and external memory such as DRAM.

During *Aggregation* phase, most vertices are not only destination vertices for their own *Aggregation* but also source vertices for their neighbors. The difference between these two roles is that, as the source vertex, the vertex feature value should stay constant until all vertices finish *Aggregation*. While as the destination vertex, the vertex feature value changes during *Aggregation*. Therefore, in the worst case, we need double storage space for every vertex if we want to avoid long-distance data movement from external DRAM. For some small-scale graphs, MRAM is big enough to do this. However, MRAM does not have enough space to store both the source and aggregated destination vertices for some middle-scale graphs. For large-scale graphs, even only storing all source vertices is difficult. We propose adaptive data flow with RS Mode and normal mode to deal with these different situations.

1) RS Mode: For some small or middle-scale graphs, the memory size of IMGA's MRAM is big enough to store all the source vertices and still has rest space. It is difficult and inefficient to store all the aggregated vertices' destination features in the rest space. In contrast, adaptive data flow works in RS Mode, setting rest space as RS and only storing vertices' destination feature in RS if necessary.

We observed that some vertices' source features would not be used later after finishing their own *Aggregation* as destination vertices. Thus, these vertices' aggregated destination feature can be written back directly to the original address of storing source feature. It is easy to prove that an aggregated destination vertex i can replace source vertex i if there is no edge to the right of the diagonal of the ith row of the adjacency matrix. As shown in Fig. 2, vertex $V3$ has no edge on the right side of the diagonal of the adjacency matrix. Therefore, $V3$'s new destination feature can be written back to the address storing the source feature before. In contrast, vertices, such as $V1$ s' destination feature values, need to be stored in RS rather than replacing source feature value directly.

After $V3$ finishes its *Aggregation*, the source feature of vertex $V1$ will never be visited. The destination feature value of

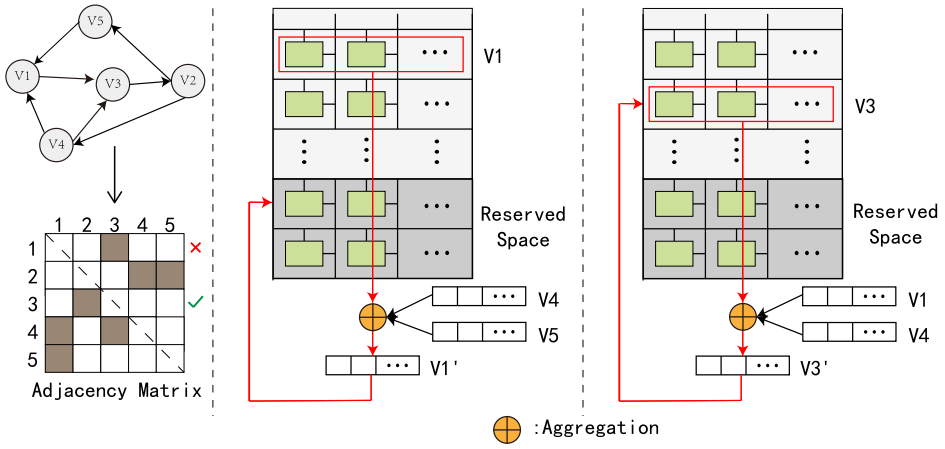

Fig. 2. Adaptive data flow in RS Mode. The vertex $V1$ and $V3$ written back to RS and original address, respectively.

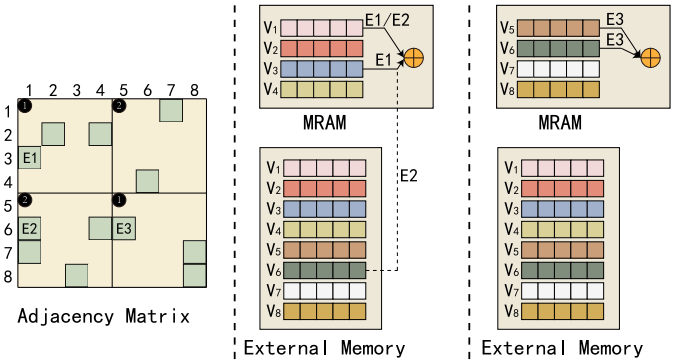

Fig. 3. Adaptive data flow in normal mode. The edges in region ❷ need to fetch vertex data from external memory, while the edges in region ❶ need not.

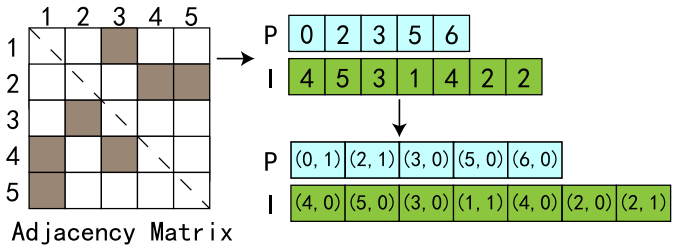

Fig. 4. Custom Edge Data Format: The topography information of graph is encoded to CSC format first and then converted to our custom edge data format with additional information used for the controller later.

$V1$ can be written back, and the RS can be used for another vertex. Thus, the total size of RS can be further reduced.

2) Normal Mode: If it is impossible to store all the source vertices in MRAM, we propose normal mode to deal with this situation. As exampled in Fig. 3, there are 8 vertices in total, and MRAM is assumed to be able to store 4 vertices simultaneously. Therefore, we have to load vertices 1–4 to MRAM first. After these vertices finish their *Aggregation*, the rest vertices will be loaded later. The RS design is not used in normal mode, and the aggregated destination vertex will be written to external DRAM. Thus, the regions of the adjacency matrix can be divided into two categories. When an edge is in region ❶, such as $E1$ and $E3$, both vertices it connects are in the MRAM so that we can perform *Aggregation* immediately. However, the source vertex of edge in region ❷ (e.g., $E2$) is not in the MRAM and needs to be read from external memory DRAM. The costly data movement between MRAM and external memory should be avoided as much as possible. We will discuss this in Section III-D.

B. Customized Edge Data Format

The employment of adaptive data flow necessitates fine-grained controls. Allowing the controller to handle complex decision-making would lead to a nontrivial hardware overhead. Nonetheless, our observations indicate that branch decisions can be made during the preprocessing phase rather than at runtime. The data flow branches are solely associated with structural graph information and are irrelevant to node properties. Consequently, we introduce a customized edge data format containing control information derived from the CSC format during the preprocessing phase.

CSC is suitable for our architecture as we simultaneously process only one destination vertex. As shown in Fig. 4, the adjacency matrix is represented by two arrays $<I, P>$. The row index array (I) is the set of row indexes of nonzero elements in the adjacency matrix. The column index pointer array (P) is the set of the number of nonzero elements in each column. However, we need more information from the adjacency matrix to support RS management in RS Mode. We need to decide whether an aggregated destination vertex needs to be written to the RS and when the vertex can be written back to its original address from the RS. We add two mask arrays $<M_I, M_P>$ for array I and P separately. When the index of array P changes from i to $i+1$, which means a destination vertex i finish its *Aggregation*, the IMGA will check its mask value in M_P. If the value equals 1, the destination vertex will be stored in RS. In the same way, when a source vertex $I(j)$ is aggregated, the index of the array I will change and the IMGA will check corresponding mask value in M_I. If the value equals 1, the vertex $I(j)$ will be removed from the RS.

LN_D denotes the last destination vertex that will aggregate the vertex i. In Algorithm 1, if the index of LN_D is larger

Algorithm 1: Generate Edge Data From the Graph's Structural Information

Data: Input graph G = (*V*,*E*), adjacency matrix of input graph $A \in \mathbb{R}^{n \times n}$, CSC edge list: pointers array *P* and indices array *I*
Result: Mask M_P for array *P* of size *l* and mask M_I for array *I*, Reserved space size *Space_total*
initialize elements of M_P and M_I as 0 and *Space_total*, *Space_now* = 0;
for $i \leftarrow 0$ **to** *l* **do**
  destination vertex index *indexD* = *i*;
  find the largest index of V_{indexD}'s neighbours LN_D;
  if $i < LN_D$ **then**
    $M_{Pi} = 1$;
    Space_now+ = 1;
    update *Space_total* = *Space_now* if *Space_now* is bigger;
  end
  for $j \leftarrow P_{i-1}$ **to** $P_i - 1$ **do**
    source vertex index *indexS* = I_j;
    find the largest index of V_{indexS}'s neighbours LN_S;
    if $LN_S <= indexD$ **then**
      $M_{Ij} = 1$;
      Space_now− = 1;
    end
  end
end

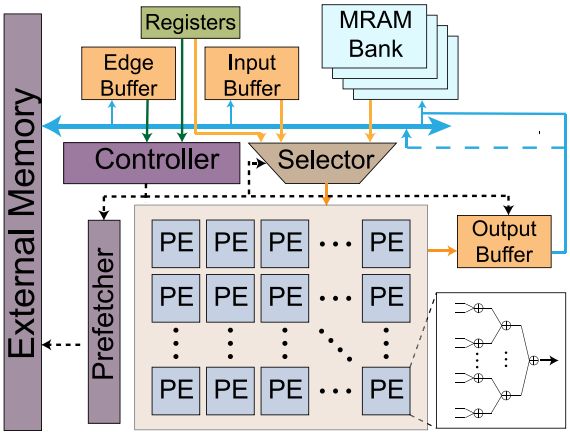

Fig. 5. Overall architecture of the proposed graph *Aggregation* acceleration engine.

than *i*, the new destination feature of vertex *i* needs to be stored in RS after *Aggregation*. If LN_D's *Aggregation* has also been done, the vertex *i* will not be used later, therefore it can be moved from the RS to the original address. As Algorithm 1 is only relevant to the adjacent matrix, which will not change during GCN computing, the algorithm only performs once for each graph. We revise the format of CSC, adding the result of Algorithm 1 to the edge data.

C. IMGA Architecture Overview

Our architecture design is demonstrated in Fig. 5. The function of components in our architecture is described as follows:

MRAM Banks: The vertex vector will be loaded in MRAM banks before *Aggregation*. Vertices are stored in MRAM in the same order as vertex indexes. In RS Mode, Each bank reserve a few rows as RS. As the total RS can be calculated from the adjacency matrix, the number of reserved rows is determined before *Aggregation*.

Input and Output Buffer: In normal mode, the source vertex read from DRAM is stored in input buffer. The aggregated destination vertex that needs to be written to DRAM is stored in Output Buffer.

Selector: The input data of adder trees may come from three different sources: 1) row buffer of a bank; 2) input buffer; and 3) zero registers. The selector is responsible for selecting the rightful source. Each bank has a selector so that the vertices in different MRAM banks can be read in parallel.

Processing Elements (PEs): As we perform *Aggregation* as vector addition between the destination vertex and its neighbors, we deploy a group of PEs near banks. A PE consists of adder trees and an accumulator. The adder tree aggregates data from different banks, and many adder trees work in parallel to aggregate different dimensions of the same vector. Accumulators are used to store the destination vertex being aggregated.

Prefetcher and Edge Buffer: Prefetcher reads edge data and missing source vertex from external memory and stores them in edge buffer and input buffer, respectively.

Registers: We deploy several registers to store the information used by controller and selector.

1) *Feature Length Register:* It is used to store the length of the feature vector.
2) *Vertices Range Register:* In normal mode, we use this register to store the upper and lower-bound indices of vertices stored in MRAM.
3) *Zero Register:* If there is no source vertex in both input buffer and MRAM Bank, the selector will choose zero registers as the input of PE.

Controller: The controller read edge data from edge buffer and convert it to control signal used in IMGA architecture. The controller converts the value of array *P* to the index of the destination vertex and maps the index to the storage address in MRAM. In the same way, the controller map the value of array *I* to the address of the source vertex. In RS Mode, the controller manages the data movement in RS by checking mask arrays in edge data. In normal mode, the controller checks the vertices range register to determine whether a source vertex is in MRAM or not. If the source vertex needs to be read from DRAM, the controller will send corresponding signal to Prefetcher. The control signal of the Selector also comes from controller. The most complicated functionality aforementioned is RS management in RS Mode. However, the customized edge data generated during the preprocessing phase provides explicit signals on whether a particular vertex should be moved into or out of the RS, thereby streamlining the controller's implementation. The controller's functionality can be realized using simple decoders that decode the input of customized edge data and value of registers into control signals, incurring minimal hardware overhead.

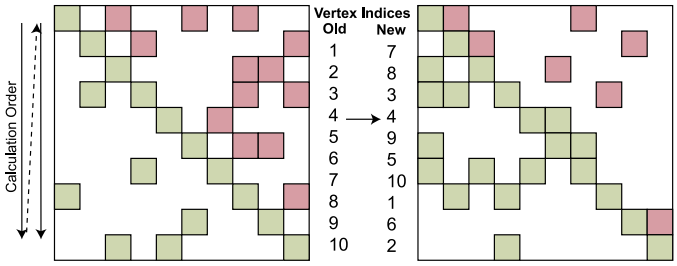

Fig. 6. Reordering optimization for RS Mode.

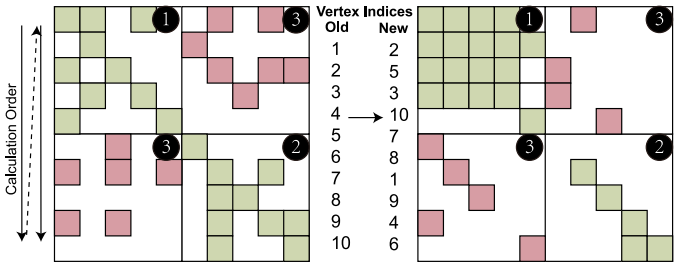

Fig. 7. Reordering optimization for normal mode.

D. Reordering Optimization

As elaborated in Section III-A, the adjacency matrix has a significant effect on the execution efficiency of IMGA. There are previous works [7], [25] point out that the overall performance of graph processing and GCN computing could be improved by reordering the vertices' index, which does not modify the topology of the graph. However, previous works mainly focus on improving cache locality or identifying clusters in the graph with sophisticated techniques, which are unsuitable for IMGA. Thus, we propose a lightweight reordering optimization for adaptive data flow, improving the performance of IMGA with hardware–software co-design without bringing huge overhead.

1) Reordering for RS Mode: As elaborated in Section III-A1, the aggregated destination vertex i can be written back to its original address directly only if there is no edge on the right side of the diagonal of the ith row in the adjacency matrix. Therefore, the target of reordering for RS Mode is to move the edge from the right side of the diagonal to the left.

As shown in Fig. 6, the *Aggregation* of vertex 8 require to keep original feature value of vertex 1, 3, 4, and 7 in MRAM until vertex 8 finish its *Aggregation*. However, if vertex 8 perform *Aggregation* first, only a few vertices' source features need to be reserved. Inspired by this, we propose a lightweight remapping algorithm. We calculate the out-degree for each vertex and sort the vertices based on that. The sorted vertices order is also the new indices. By doing this, vertices in the lower order are less dependent on the vertices in the earlier order. Thus, the possibility of a vertex can be written back to the original address increase.

2) Reordering for Normal Mode: For normal mode, we can still use the reordering technique to improve the performance. As shown in Fig. 7, IMGA process the edges in diagonal blocks of adjacency matrix (❶ and ❷) efficiently because both source and destination vertices are in MRAM. In contrast, the source vertex of edge in region ❸ needs to be read from DRAM. Reordering aims to make the edges converge toward the diagonal blocks. The nature graph obeys power-law degree distribution [26], which means most vertices cluster around a few high-degree vertices. According to this feature, we can make the indexes of the same cluster vertices continuous by reordering. If a whole cluster of vertices is loaded in MRAM, there will be a lower possibility of reading source vertex out of the cluster. However, it is costly to recognize the clusters of a graph precisely. Therefore, we propose a lightweight algorithm to achieve approximate effects. There is a large proportion of edges between a few rich vertices. The principle of the reordering algorithm is to calculate both out-degree and in-degree for each vertex and sort the vertices based on them. Thus, we can ensure the edges between high-degree vertices can be moved to a diagonal block at least.

TABLE II
DATASET INFORMATION

Datasets	Vertex	Edge	Feature Length
Cora(CR)	2,708	10,556	1,433
Pubmed(PB)	19,717	88,648	500
Citeseer(CS)	3,327	9,104	3,703
Ogbn-arxiv(OA)	169,343	1,166,243	128
Ogbl-collab(OC)	235,868	2,358,104	128
Ogbl-ppa(OP)	576,289	42,463,862	58
Reddit(RD)	232,965	114,615,892	602

IV. EVALUATION

A. Experimental Setup

We evaluate the performance of IMGA by simulating its behavior based on device parameters. We first simulate IMGA's execution time and energy cost of performing *Aggregation* on popular graph datasets shown in Table II. The datasets CR, PB, and CS are selected as small-scale graphs, and OA, OC, OP, and RD are selected as middle and large-scale graphs in our experiments. The parameter of SOT-MRAM is obtained from [27], and the parameter of the adder is obtained from [28]. Due to limitations in current density, the MRAM capacity is restricted, and a capacity parameter of 32 MB is chosen as it is a reasonable and moderate value for MRAM. The PE's frequency is decided by the critical path of IMGA, which consists of the latency of memory reading, the multiplexer and the adders in the adder tree. The number of PEs is relative to the number of MRAM banks and the bank's row buffer size. To compare our work with the previous PIM-based GCN accelerator, we simulate the computing time of complete GCN by integrating the V100 GPU as a *Combination* engine and connecting IMGA and GPU by PCI-E bus.

Then, we analyze the effect of our optimization strategies, such as RS design and reordering for adaptive data flow. To compare IMGA with CPU and GPU, we also implement *Aggregation* by state-of-the-art software framework Pytorch Geometric [29] on Intel Xeon E5-2620 CPU and V100 GPU.

TABLE III
CONFIGURATION OF BASELINE AND IMGA

	CPU	GPU	IMGA
Compute Unit	2.1GHz @ 16 cores	1.25GHz @ 5120 cores	333 MHz @ 128PE
On-chip Memory	40MB SRAM	34MB SRAM	32MB MRAM
Off-chip Memory	136.6GB/s DDR4	900GB/s HBM 2.0	136.6GB/s DDR4

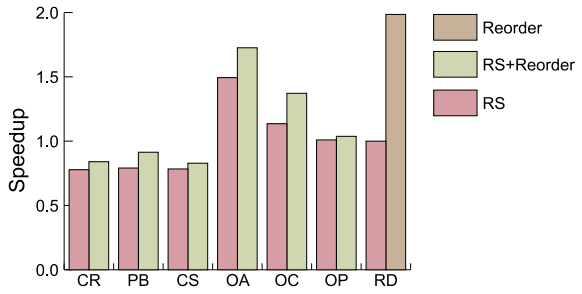

Fig. 8. Results of reordering optimization for adaptive data flows.

The detailed configuration of CPU and GPU is shown in Table III.

B. Overall Results

1) Benefits of Data Flow Optimization: We evaluate the performance of IMGA with reordering optimization. As shown in Fig. 8, the performance of IMGA has decreased slightly on datasets CR∼CS. The reason is that these datasets are small enough to store both destination vertices and source vertices in MRAM. The RS design needs data movement between RS and normal space, which brings extra cost from them. For datasets OA∼OP, MRAM has space to store all the source vertices and part of the destination vertex. The rest of the destination vertices have to be stored in DRAM. The reserve space design reduces the long-distance data movement between DRAM and MRAM, therefore the execution time of IMGA decreases significantly. Only on dataset RD, it is difficult for MRAM of IMGA to store source vertices, and RS design has no use for dataset RD. IMGA works in normal mode in this situation. The target of reordering optimization for normal mode is to put more computation into MRAM. After reordering optimization, most *Aggregation* computing will not need to read vertex data from DRAM.

We further analyze the effect of our optimization on the access of DRAM and the time breakdown of our architecture. As shown in Fig. 9, for small datasets, such as CR, most of the time is spent in MRAM. There are still some time costs in DRAM because we need to read edge data from DRAM. For middle-scale datasets, such as OP, part of aggregated destination vertices have to be stored in DRAM, therefore the time cost in DRAM increases. Our RS design and reordering optimization reduce the storage space of aggregated destination vertices, decreasing the time cost in DRAM. On dataset RD, most of the time is spent in accessing DRAM.

Fig. 9. Time breakdown of IMGA with and without the optimization of RS and reorder. Baseline represents IMGA without RS and reordering.

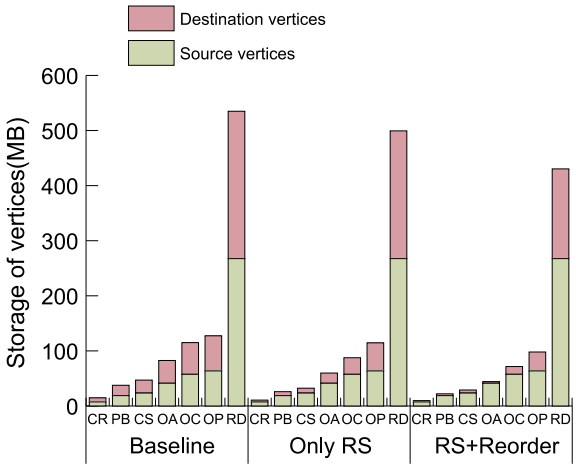

Fig. 10. Effect of RS and reordering optimization on storage cost of vertices.

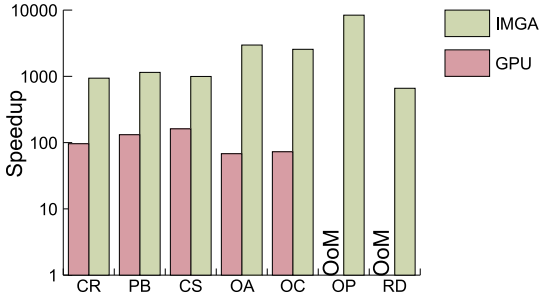

Fig. 11. Runtime comparison normalized to CPU. OoM denotes out of memory.

Besides, we evaluate the reserve space design with and without the reordering optimization. As shown in Fig. 10, the reserve space design needs about 27.1%–46.4% of the total space to store aggregated destination vertex. After reordering optimization, 14.8%–37.8% total space is needed as RS. Reserve space design and reordering optimization reduce the space needed to store the destination vertices significantly.

2) Speedup: Fig. 11 depicts that IMGA achieves an average 2523× speedup compared with CPU and 21× speedup compared with GPU. The performance improvement comes from several aspects. First, IMGA utilizes the bank-level high

TABLE IV
SPEEDUP OF IMGA+ AND OTHER PIM-BASED ACCELERATORS OVER GPU

Datasets	DCIM-GCN[30]	TARe[19]	IMGA+
CR	109×	36×	48×
PB	29×	120×	12×
CS	48×	22×	49×
OA	-	-	19×
OC	-	-	8×
OP	-	-	10×
RD	-	-	7×

Note: The IMGA+ represents the IMGA integrated with GPU. The symbol "-" means the architecture did not evaluate its performance on the dataset.

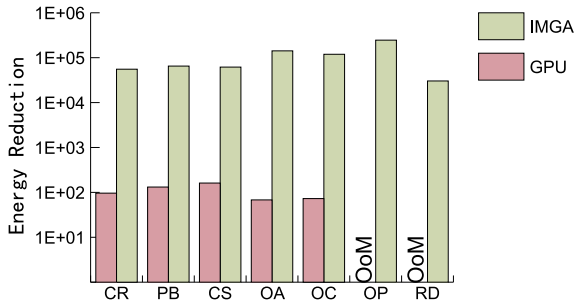

Fig. 12. Energy reduction comparison normalized to CPU. OoM denotes out of memory.

bandwidth inside the MRAM sufficiently. Second, improvement of vertices data reuse makes the *Aggregation* computing performed where the data stored and the reduction of long-distance data movement effectively reduces the time cost of reading data from DRAM. Third, proposed optimizations, such as RS design and reordering, reduce the I/O cost significantly.

We use IMGA+ represents the IMGA integrated with GPU. Table IV shows the speedup of IMGA+ and the other PIM-based GCN accelerators over GPU when evaluating complete GCN flow. We select the state-of-the-art work DCIM-GCN [30], and TARe [19] for experiment. We quantized the GCN model to fixed point format with 8-bit activation and 8-bit weight. As the GPU has huge computing power and is suitable for *Combination*, which essentially is dense matrix-vector multiplication (MVM), the computation latency is still dominated by *Aggregation* phase in IMGA.

As the RS and reordering optimization take effect for middle and large-scale graphs, the performance of IMGA+ is comparable to that of other accelerators on small graphs, such as CR, PB, and CS, which is consistent with the experiment result in Fig. 8. Due to the lack of experimental data of TARe and DCIM-GCN on other datasets, the comparison results of the relatively large graphs cannot be presented.

3) Energy Cost: Fig. 12 shows that our architecture achieves 1.03E+06× energy reduction over CPU and 1.04E+03× energy reduction over GPU. Energy consumption reduction comes from three aspects. First, *Aggregation* is performed near memory, and we propose adaptive data flow with optimization. Therefore, energy consumed in long-distance data movement can be saved significantly. Second, MRAM has a lower-read and write energy cost compared to SRAM and DRAM used in CPU and GPU. As a nonvolatile memory (NVM), there is also no need to refresh data frequently like DRAM. Third, IMGA's acceleration on *Aggregation* decreases execution time significantly, which causes a reduction in static power consumption.

TABLE V
ENERGY REDUCTION OF IMGA+ AND OTHER PIM-BASED ACCELERATORS OVER GPU

Datasets	DCIM-GCN[30]	TARe[19]	IMGA+
CR	36574×	29863×	672×
PB	9781×	8200×	296
CS	16054×	19663×	366×
OA	-	-	611×
OC	-	-	360×
OP	-	-	687×
RD	-	-	910×

Table V shows energy reduction comparisons of IMGA+ and other accelerators over GPU. The energy reduction achieved by IMGA+ is not superior to that of other approaches. The primary source of energy overhead in IMGA+ comes from the GPU and data transportation between GPU and IMGA. However, as the *Combination* engine is optional and replaceable, the issue of high-energy cost can be solved by using other power-efficiency accelerators as *Combination* engine, such as a PIM accelerator based on other types of memory.

V. DISCUSSION

In this section, we discuss about the application of IMGA in GCN and compare IMGA with other PIM architecture. In Section V-A, we discuss about how the IMGA architecture integrates with *Combination* engine for GCN computing. In Section V-B, we discuss about the comparison between IMGA and other PIM architecture based on different kinds of memory, such as hybrid memory cube (HMC) and ReRAM.

A. Application in GCN

IMGA concentrates on accelerating *Aggregation* because it is the memory access bottleneck of GCN. However, GCN performs both *Aggregation* and *Combination* for input graph data, thus we consider how to apply IMGA in complete GCN computing. The *Combination* operation involves a MVM that is computationally intensive, which has already been extensively researched regrading its acceleration. Thus, an alternative computing engine, such as a CPU, GPU, or other memory-based PIM architecture, can be integrated to perform the *Combination* phase. We have presented experimental results of the IMGA integrated with GPU. Here, we further discuss about a possible three-level memory architecture design when the *Combination* engine is integrated. The three-level memory architecture consists of DRAM, MRAM, and cache. The quantity of feature vectors transferred from IMGA to the combination engine in a single GCN layer is the number of vertices, considerably less than the number of edges in the aggregation phase. Therefore, the vertex feature should be sent to MRAM from DRAM first and perform

Aggregation in MRAM. Vertices that have been aggregated will be written to the cache where the *Combination* engine can fetch data. MRAM has to wait for the *Combination* engine because the combined result is new input data of the next iteration *Aggregation*. A vertex can be written to cache as soon as it has been aggregated rather than waiting for all vertexes to be aggregated to reduce waiting time. Note that *Combination* can be implemented with a CPU or other PIM process engine, and the three-level memory architecture should be modified if the GPU is used as a *Combination* engine because it accesses data from DRAM directly by DMA. In this case, the cache is replaced with the swap space in DRAM.

B. Comparison With Other PIM Architecture

Many PIM-based accelerators have been proposed over the past few years. They can be divided into several categories according to the type of memory used. The SRAM-based PIM has undergone extensive research. Even though SRAM has fast write/read speed and mature manufacture, the low density of SRAM limits its application in the GCN computation of large graphs. high-bandwidth memory (HBM)/HMC-based PIM accelerators [12], [31], [32], [33], [34], [35] are relatively close to practical because the HBM and HMC are commercially mature. HBM has been integrated wildly with the GPU. Compared to HBM, HMC draw more attention in the PIM field for its intercube communication mechanism. However, both of them are volatile memory devices and have higher-energy costs for computing and reading/writing data than NVM. Besides, it is challenging to deploy complex computing logic in these mature memory.

The NVM-based PIM accelerator has drawn much attention recently. We choose MRAM as our memory device of PIM architecture for its low-energy cost and fast data access speed. There are also some ReRAM-based PIM architectures. Compared with MRAM, ReRAM can store multibit values in one cell, which makes ReRAM suitable for performing multibit multiplication in ReRAM crossbar [36]. For example, the *Combination* phase can be performed in ReRAM. However, the endurance of ReRAM limits its application in the big-data field [27]. Besides, the ability of multibit multiplication storage is not necessary for our architecture as we treat *Aggregation* as vector addition, and we perform computing near the memory cells rather than in them.

Despite the advantages of low-energy consumption compared with HBM and HMC, NVM generally suffers from low density for its immature in commercial. Thus, the capacity of NVM is much smaller than DRAM currently. For example, the MRAM used in our architecture only stores 32-MB data. However, the real-world graph can be large-scale and it is difficult for NVM-based PIM architecture to process gigabyte data on chip. But if PIM architecture has to access data from an external memory, such as DRAM frequently, the PIM architecture may degenerate into a non-PIM accelerator. According to our observation, most NVM-PIM GCN accelerators choose to ignore this problem temporary. In IMGA, we propose many techniques to reduce the data movement between MRAM and DRAM. With the development of NVM technology, the density and capacity of MRAM will improve in the future. But the scale of graph data is also growing. Therefore, we believe it is necessary to pay more attention to solving this problem from the aspect of architectural design.

VI. Conclusion

GCN *Aggregation* requires frequent and random memory access, which causes low-performance and energy-inefficient in traditional architecture like CPU and GPU. In this article, we propose an in-memory graph *Aggregation* acceleration architecture IMGA to realize in-situ *Aggregation*. The vertices data are stored in SOT-MRAM arrays, and the edge data is utilized as a control signal for automatic feature vector addition. We propose adaptive data flow for different scale graphs to reduce the long-distance data movement between the PIM accelerator and external memory. We also design customized edge data to take further advantage of the graph's structural information and support the control flow of our architecture. The reordering optimization is proposed to improve the performance of IMGA from the aspect of hardware–software co-design. We demonstrate that the proposed *Aggregation* acceleration engine IMGA can be applied in a wide range of graph datasets. Experimental results demonstrate promising speedups and energy efficiency.

References

[1] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021.

[2] M. Yan et al., "HyGCN: A GCN accelerator with hybrid architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2020, pp. 15–29.

[3] J. Lin, S. Li, Y. Ding, and Y. Xie, "Overcoming the memory hierarchy inefficiencies in graph processing applications," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, 2021, pp. 1–9.

[4] S. Liang et al., "EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Trans. Comput.*, vol. 70, no. 9, pp. 1511–1525, Sep. 2021.

[5] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *Proc. 57th ACM/IEEE Des. Automat. Conf. (DAC)*, 2020, pp. 1–6.

[6] T. Geng et al., "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2020, pp. 922–936.

[7] T. Geng et al., "I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2021, pp. 1051–1063.

[8] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2018, pp. 531–543.

[9] S. Angizi, J. Sun, W. Zhang, and D. Fan, "GraphS: A graph processing accelerator leveraging SOT-MRAM," in *Proc. Des. Automat. Test Europe Conf. Exhib. (DATE)*, 2019, pp. 378–383.

[10] S. Angizi and D. Fan, "GraphiDe: A graph processing accelerator leveraging in-DRAM-computing," in *Proc. Great Lakes Symp. VLSI*, 2019, pp. 45–50.

[11] 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 *Proc. 24th Asia South Pac. Des. Automat. Conf.*, 2019, pp. 120–126.

[12] G. Dai et al., "GraphH: A processing-in-memory architecture for large-scale graph processing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 4, pp. 640–653, Apr. 2019.

[13] X. Wang et al., "Triangle counting accelerations: From algorithm to in-memory computing architecture," *IEEE Trans. Comput.*, vol. 71, no. 10, pp. 2462–2472, Oct. 2022.

[14] X. Chen, X. Wang, X. Jia, J. Yang, G. Qu, and W. Zhao, "Accelerating graph-connected component computation with emerging processing-in-memory architecture," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 12, pp. 5333–5342, Dec. 2022.
[15] J. Chen, G. Lin, J. Chen, and Y. Wang, "Towards efficient allocation of graph convolutional networks on hybrid computation-in-memory architecture," *Sci. China Inf. Sci.*, vol. 64, no. 6, pp. 1–14, 2021.
[16] Z. Wang et al., "GNN-PIM: A processing-in-memory architecture for graph neural networks," in *Proc. Conf. Adv. Comput. Archit.*, 2020, pp. 73–86.
[17] T. Yang et al., "PIMGCN: A ReRAM-based PIM design for graph convolutional network acceleration," in *Proc. 58th ACM/IEEE Des. Automat. Conf. (DAC)*, 2021, pp. 583–588.
[18] N. Challapalle, K. Swaminathan, N. Chandramoorthy, and V. Narayanan, "Crossbar based processing in memory accelerator architecture for graph convolutional networks," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, 2021, pp. 1–9.
[19] Y. He, Y. Wang, C. Liu, H. Li, and X. Li, "TARe: Task-adaptive in-situ ReRAM computing for graph learning," in *Proc. 58th ACM/IEEE Des. Automat. Conf. (DAC)*, 2021, pp. 577–582.
[20] H. Wang, W. Kang, B. Pan, H. Zhang, E. Deng, and W. Zhao, "Spintronic computing-in-memory architecture based on voltage-controlled spin–orbit torque devices for binary neural networks," *IEEE Trans. Electron Devices*, vol. 68, no. 10, pp. 4944–4950, Oct. 2021.
[21] Z. Guo et al., "Spintronics for energy-efficient computing: An overview and outlook," *Proc. IEEE*, vol. 109, no. 8, pp. 1398–1417, Aug. 2021.
[22] J. Yang et al., "Exploiting spin–orbit torque devices as reconfigurable logic for circuit obfuscation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 1, pp. 57–69, Jan. 2019.
[23] Y. Zhao et al., "NAND-SPIN-based processing-in-MRAM architecture for convolutional neural network acceleration," 2022, *arXiv:2204.09989*.
[24] S.-W. Chung et al., "4Gbit density STT-MRAM using perpendicular MTJ realized with compact cell structure," in *Proc. IEEE Int. Electron Devices Meeting (IEDM)*, 2016, pp. 1–4.
[25] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, 2018, pp. 203–214.
[26] 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. Oper. Syst. Des. Implement. (OSDI)*, 2012, pp. 17–30.
[27] S. Angizi et al., "Accelerating deep neural networks in processing-in-memory platforms: Analog or digital approach?" in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, 2019, pp. 197–202.
[28] T. Yang, T. Ukezono, and T. Sato, "A low-power configurable adder for approximate applications," in *Proc. 19th Int. Symp. Qual. Electron. Des. (ISQED)*, 2018, pp. 347–352.
[29] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric," 2019, *arXiv:1903.02428*.
[30] Y. Qiu, Y. Ma, W. Zhao, M. Wu, L. Ye, and R. Huang, "DCIM-GCN: Digital computing-in-memory to efficiently accelerate graph convolutional networks," in *Proc. 41st IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2022, pp. 1–9.
[31] Y.-C. Kwon et al., "25.4 A 20nm 6GB function-in-memory DRAM, based on HBM2 with a 1.2TFLOPS programmable computing unit using bank-level parallelism, for machine learning applications," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, vol. 64, 2021, pp. 350–352.
[32] S. Kim et al., "Processing-in-memory in high bandwidth memory (PIM-HBM) architecture with energy-efficient and low latency channels for high bandwidth system," in *Proc. IEEE 28th Conf. Elect. Perform. Electron. Packag. Syst. (EPEPS)*, 2019, pp. 1–3.
[33] Y. Zhuo et al., "GraphQ: Scalable PIM-based graph processing," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 712–725.
[34] X. Xie et al., "SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2021, pp. 570–583.
[35] Z. Li, X. Chen, and Y. Han, "Optimal data allocation for graph processing in processing-in-memory systems," in *Proc. 27th Asia South Pac. Des. Autom. Conf. (ASP-DAC)*, 2022, pp. 238–243.
[36] C. Xu et al., "Overcoming the challenges of crossbar resistive memory architectures," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2015, pp. 476–488.

Yuntao Wei received the B.S. degree in computer science and technology from Dalian University of Technology, Dalian, China, in 2020. He is currently pursuing the M.S. degree with the School of Integrated Circuit Science and Engineering, Beihang University, Beijing, China.

His research interests include the graph computing accelerations with emerging in-memory computing architectures.

Xueyan Wang (Member, IEEE) received the B.S. degree in computer science and technology from Shandong University, Jinan, China, in 2013, and the Ph.D. degree in computer science and technology from Tsinghua University, Beijing, China, in 2018.

From 2015 to 2016, she was a Visiting Scholar with the University of Maryland at College Park, College Park, MD, USA. She is currently an Assistant Professor with the School of Integrated Circuit Science and Engineering, Beihang University, Beijing, China. Her current research interests include processing-in-memory architectures, AI chip, and hardware security.

Shangtong Zhang received the B.S. degree in computer science and technology from Dalian University of Technology, Dalian, China, in 2020. He is currently pursuing the M.S. degree with the School of Integrated Circuit Science and Engineering, Beihang University, Beijing, China.

His research interests include the graph computing accelerations with emerging in-memory computing architectures.

Jianlei Yang (Senior Member, IEEE) received the B.S. degree in microelectronics from Xidian University, Xi'an, China, in 2009, and the Ph.D. degree in computer science and technology from Tsinghua University, Beijing, China, in 2014.

He is currently an Associate Professor with the School of Computer Science and Engineering, Beihang University, Beijing. From 2014 to 2016, he was a Postdoctoral Researcher with the Department of ECE, University of Pittsburgh, Pittsburgh, PA, USA. His current research interests include computer architectures and neuromorphic computing systems.

Dr. Yang was the recipient of the First/Second Place on ACM TAU Power Grid Simulation Contest in 2011/2012. He was a recipient of the IEEE ICCD Best Paper Award in 2013, the ACM GLSVLSI Best Paper Nomination in 2015, the IEEE ICESS Best Paper Award in 2017, and the ACM SIGKDD Best Student Paper Award in 2020.

Xiaotao Jia (Member, IEEE) received the B.S. degree in mathematics from Beijing Jiao Tong University, Beijing, China, in 2011, and the Ph.D. degree in computer science and technology from Tsinghua University, Beijing, in 2016.

He is currently an Associate Professor with the School of Microelectronics, Beihang University, Beijing, where he was a Postdoctoral Researcher in Microelectronics, from 2016 to 2019. His current research interests include spintronic circuits, stochastic computing, and Bayesian deep learning.

Zhaohao Wang (Senior Member, IEEE) received the B.S. degree in microelectronics from Tianjin University, Tianjin, China, in 2009, the M.S. degree in microelectronics from Beihang University, Beijing, China, in 2012, and the Ph.D. degree in physics from the University of Paris-Saclay, Paris, France, in 2015.

He is currently an Associate Professor with the School of Integrated Circuit Science and Engineering, Beihang University. His current research interests include the modeling of nonvolatile nanodevices and the design of emerging nonvolatile memories and logic circuits.

Gang Qu (Fellow, IEEE) received the B.S. and M.S. degrees in mathematics from the University of Science and Technology of China, Hefei, China, in 1992 and 1994, respectively, and the Ph.D. degree in computer science from the University of California, Los Angeles, Los Angeles, California, in 2000.

Upon graduation, he joined the University of Maryland at College Park, College Park, MD, USA, where he is currently a Professor with the Department of Electrical and Computer Engineering and the Institute for Systems Research. He is the Director of Maryland Embedded Systems and Hardware Security Lab and the Wireless Sensors Laboratory. His primary research interests are in the areas of embedded systems and VLSI CAD with focus on low power system design and hardware related security and trust.

Dr. Qu has served 18 times as the general or program chair/co-chair for conferences, symposiums, and workshops. He is the Co-Founder of IEEE Asian Hardware Oriented Security and Trust Symposium, Top Picks in Hardware and System Security Workshop, and the IEEE CEDA Hardware Security and Trust Technical Committee.

Weisheng Zhao (Fellow, IEEE) received the Ph.D. degree in physics from the University of Paris Sud, Paris, France, in 2007.

He is currently the Professor with the School of Integrated Circuit Science and Engineering, Beihang University, Beijing, China. In 2009, he joined the French National Research Center, as a Tenured Research Scientist. Since 2014, he has been a Distinguished Professor with Beihang University. He has published more than 200 scientific articles in leading journals and conferences, such as *Nature Electronics*, *Nature Communications*, *Advanced Materials*, IEEE TRANSACTIONS, ISCA, and DAC. His current research interests include the hybrid integration of nanodevices with CMOS circuit and new nonvolatile memory (40-nm technology node and below) like MRAM circuit and architecture design.

Prof. Zhao is currently the Editor-in-Chief for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS I: REGULAR PAPER.