Triangle Counting Accelerations: From Algorithm to In-Memory Computing Architecture

Xueyan Wang, *Member, IEEE*, Jianlei Yang, *Senior Member, IEEE*, Yinglin Zhao, Xiaotao Jia, Rong Yin, Xuhang Chen, Gang Qu, *Fellow, IEEE*, and Weisheng Zhao, *Fellow, IEEE*

Abstract—Triangles are the basic substructure of networks and triangle counting (TC) has been a fundamental graph computing problem in numerous fields such as social network analysis. Nevertheless, like other graph computing problems, due to the high memory-computation ratio and random memory access pattern, TC involves a large amount of data transfers thus suffers from the bandwidth bottleneck in the traditional Von-Neumann architecture. To overcome this challenge, in this paper, we propose to accelerate TC with the emerging processing-in-memory (PIM) architecture through an algorithm-architecture co-optimization manner. To enable the efficient in-memory implementations, we come up to reformulate TC with bitwise logic operations (such as AND), and develop customized graph compression and mapping techniques for efficient data flow management. With the emerging computational Spin-Transfer Torque Magnetic RAM (STT-MRAM) array, which is one of the most promising PIM enabling techniques, the device-to-architecture co-simulation results demonstrate that the proposed TC in-memory accelerator outperforms the state-of-the-art GPU and FPGA accelerations by $12.2\times$ and $31.8\times$, respectively, and achieves a $34\times$ energy efficiency improvement over the FPGA accelerator.

Index Terms—Triangle counting acceleration, processing-in-memory, algorithm-architecture co-design, graph computing

1 Introduction

Triangle counting (TC) counts the number of triangles in a given graph and it is an basic problem in graph computing. TC problem is not hard but it is memory bandwidth intensive thus time-consuming. As a result, researchers from both academia and industry have proposed many TC acceleration methods ranging from sequential to parallel, single-machine to distributed, and exact to approximate. From the computing hardware perspective, these acceleration strategies are generally executed on CPU, GPU or FPGA, and are based on Von-Neumann architecture [1], [2], [3]. However, due to the fact that most graph processing algorithms have low computation-memory ratio and high random data access patterns, there are frequent data transfers between the computational unit and memory components which consumes a large amount of time and energy, the existing acceleration approaches can only alleviate by parallelism while cannot resolve the bottleneck of data transfers.

Through performing computation where the data resides, in-memory computing paradigm can save most of the off-chip data communication energy and latency by exploiting the large internal memory inherent bandwidth and inherent parallelism [4], [5]. As a result, in-memory computing has appeared as a viable way to carry out the computationally-expensive and memory-intensive tasks [6], [7]. This becomes even more promising when being integrated with the emerging non-volatile Spin-Transfer Torque Magnetic RAM (STT-MRAM) memory technologies. This integration offers fast write speed, low write energy, and high write endurance among many other benefits [8], [9].

However, compared to the traditional Von-Neumann computing architecture, in which the CPU has very powerful and complex computing capabilities and control capabilities, the relatively dispersed in-memory processing cores in the spin-based in-memory computing architecture are more suitable for processing tasks that has relatively simple types of calculations and simple control logic. Due to such data transmission mode and computing characteristics of the in-memory computing architecture, traditional graph algorithms are often not well applied to in-memory computing. In the literature, there have been some explorations on in-memory graph algorithm accelerations [10], [11], [12], [13]. As analyzed above, existing TC algorithms cannot be efficiently implemented in memory.

- *Xueyan Wang, Yinglin Zhao, Xiaotao Jia, Xuhang Chen, and Weisheng Zhao are with the MIIT Key Laboratory of Spintronics, School of Integrated Circuit Science and Engineering, Beihang University, Beijing 100191, China. E-mail: {wangxueyan, jiaxt, weisheng.zhao}@buaa.edu.cn, wssdzyl@sina.com, 605003522@qq.com.*
- *Jianlei Yang is with the School of Computer Science and Engineering, BDBC, State Key Laboratory of Software Development Environment (NLSDE), Beihang University, Beijing 100191, China. E-mail: jianlei@buaa.edu.cn.*
- *Rong Yin is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100049, China. E-mail: yinrong@iie.ac.cn.*
- *Gang Qu is with the Department of Electrical and Computer Engineering and Institute for Systems Research, University of Maryland, College Park, MD 20742 USA. E-mail: gangqu@umd.edu.*

Manuscript received 26 Mar. 2021; revised 22 July 2021; accepted 22 Sept. 2021. Date of publication 26 Nov. 2021; date of current version 8 Sept. 2022.
This work of Xueyan Wang was supported in part by the National Natural Science Foundation of China under Grant 62004011 and in part by the State Key Laboratory of Computer Architecture under Grant CARCH201917. 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 by the Joint Funds of the National Natural Science Foundation of China under Grant U20A20204. The work of Rong Yin was supported in part by the Special Research Assistant Project of CAS under Grant E0YY221-2020000702 and in part by the National Natural Science Foundation of China under Grant 62106259.
(Corresponding author: Jianlei Yang and Weisheng Zhao.)
Recommended for acceptance by A. Coskun.
Digital Object Identifier no. 10.1109/TC.2021.3131049

For example, the intersection-based ones cannot be directly implemented in memory, and the matrix multiplication-based ones involve complex arithmetic computations which require non-trivial design overheads while implemented in-memory. In addition, for large sparse graphs, efficient graph data compression and data mapping mechanisms are all critical for PIM accelerations. The existing data compression methods for sparse graph, such as compressed sparse column (CSC), compressed sparse row (CSR), and coordinate list (COO) [10], cannot be directly applied to in-memory computation either.

In this paper, we propose and design the first triangle counting in-memory accelerator, called TCIM, that overcomes the above barriers through an algorithm-architecture co-optimization approach. We find that the number of triangles in a given graph can be computed using only `AND` and `BitCount` operations. Once the problem has been framed in this form, it can be efficiently implemented in an in-memory manner. The contributions of this paper can be summarized as follows.

- A hardware-friendly triangle counting method is proposed using bitwise logic operations. Such reformulation of triangle counting is amenable to in-memory implementations.
- We propose customized data slicing for efficient graph data compression, and graph data flow management strategies for mapping onto in-memory computation architectures.
- To support in-memory TC accelerations, a sparsity-aware processing-in-memory architecture is proposed utilizing state-of-the-art STT-MRAM technology. We also develop a device-to-architecture co-simulation framework for validating the proposed strategies.

The rest of the paper is organized as follows. Section 2 provides some preliminary knowledge of triangle counting and in-memory computing. Section 3 introduces the proposed TC method with bitwise operations, and Section 4 elaborates sparsity-aware data management strategies. Section 5 introduces the overall PIM architecture. Section 6 demonstrates the experimental results and Section 7 concludes the paper.

2 Preliminary

2.1 Triangle Counting

Triangle counting problem seeks to determine the number of triangles in a given graph. It is essential for analyzing networks and generally considered as the first fundamental step in calculating metrics such as clustering coefficient and transitivity ratio, as well as other tasks such as community discovery, link prediction, and Spam filtering [1]. For example, the commonly used social analysis algorithm, community discovery, gives the number of triangles in a social network to analyze which circles are more stable and have closer relationships. For a person's social circle, the more triangles there are, the stronger and closer his social relationship is. For network science in biology and neuroscience, it is also found useful to demonstrate the self-optimization phenomenon in brain's neuronal networks [14] and to control biological network [15]. The sequential algorithms for TC can be classified into two groups.

In the matrix multiplication based algorithms, a triangle is a closed path of length three, namely a path of three vertices begins and ends at the same vertex. If $\mathbf{A}$ is the adjacency matrix of graph G, $\mathbf{A}^3[i][i]$ represents the number of paths of length three beginning and ending with vertex i. Given that a triangle has three vertices and will be counted for each vertex, and the graph is undirected (that is, a triangle $i-p-q-i$ will be also counted as $i-q-p-i$), the number of triangles in G can be obtained as $trace(\mathbf{A}^3)/6$, where $trace$ is the sum of elements on the main diagonal of a matrix.

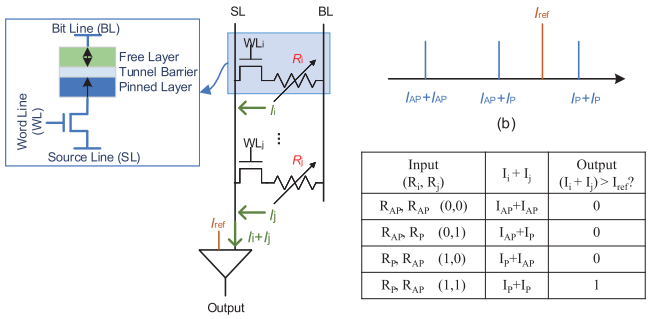
In the set intersection based algorithms, it iterates over each edge and finds common elements from adjacency lists of head and tail nodes. A lot of CPU, GPU and FPGA based optimization techniques have been proposed [1], [2], [3]. These works show promising results of accelerating TC, however, these strategies all suffer from the performance and energy bottlenecks brought by the significant amount of data transfers in TC.

2.2 In-Memory Computing With STT-MRAM

In-Memory Computing efforts can be classified into two categories according to whether they target at application-specific computations [16], [17], [18] or general-purpose computations [5], [9], [19], [20], [21], [22], [23]. ReRAM has been extensively explored and used to implement matrix-vector multiplication for neural network accelerations, with the multi-bit storage property. Comparatively, STT-MRAM has higher write endurance, faster write speed, lower write energy, while it only has limited resistance difference between the distinct resistance states of MTJ [8]. In particular, prototype STT-MRAM chip demonstrations and commercial MRAM products have been available by companies such as Everspin and TSMC. As a result, STT-MRAM is widely used to implement bit-wise boolean operations for general-purpose in-memory computing paradigm [9], [24]. In this paper, we focus on such general-purpose PIM, which can be widely used in various categories of applications.

STT-MRAM stores data with magnetic-resistances instead of conventional charge based store and access. Due to this current sensing mechanism in STT-MRAM and the fact that current can be accumulated, STT-MRAM is able to realize logic functions conveniently. This enables MRAM to provide inherent computing capabilities for bitwise logic with the core bit-cell and array structure of STT-MRAM remain unchanged, and only needs minor changes to peripheral circuitry (such as sensing circuitry to generate required sensing current) [9][25].

As Fig. 1a shows, a typical STT-MRAM bit-cell consists of an access transistor and a Magnetic Tunnel Junction (MTJ), which is controlled by bit-line (BL), word-line (WL) and source-line (SL). The relative magnetic orientations of pinned ferromagnetic layer (PL) and free ferromagnetic layer (FL) 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. The `READ` operation is done by enabling WL signal, applying a voltage V_{read} across BL and SL, and sensing the current (I_{P} or I_{AP}) that flows through the MTJ. By comparing the sense current with a reference current (I_{ref},), the data stored in a MTJ cell (logic '0' or logic '1') could be readout. The `WRITE` operation can be performed by enabling WL, then applying an appropriate voltage (V_{write}) across BL and SL to pass a current that is greater than the critical MTJ switching current. To perform bitwise logic operation, by simultaneously enabling WL_{i} and WL_{j}, then applying V_{read} across



Input (R_i, R_j)	$I_i + I_j$	Output ($I_i + I_j > I_{ref}$?)
R_{AP}, R_{AP} (0,0)	$I_{AP}+I_{AP}$	0
R_{AP}, R_P (0,1)	$I_{AP}+I_P$	0
R_P, R_{AP} (1,0)	I_P+I_{AP}	0
R_P, R_{AP} (1,1)	I_P+I_P	1

Fig. 1. An overview of performing Boolean AND operation with STT-MRAM. (a) Typical STT-MRAM bit-cells and computing paradigm. (b) The reference current. (c) Truth table.

BL and SL, the current that feeds into the sense amplifier (SA) is a summation of $I_{\rm i} + I_{\rm j}$. With different reference sensing current, various logic functions of the enabled word line can be implemented. For example, as shown in Fig. 1b, when $I_{ref} \in (I_{\rm AP} + I_{\rm P}, I_{\rm P} + I_{\rm P})$, the truth table is demonstrated in Fig. 1c, corresponds to AND logic.

Fig. 2 demonstrates the STT-MRAM arrays that support in-memory logic computations. By simultaneously enabling word-line $\rm WL_i$ and $\rm WL_j$, then applying V_{read} across $\rm BL_k$ and $\rm SL_k$ ($\rm k \in [0, n-1]$), the current that feeds into the kth SA is a summation of the currents flowing through $MTJ_{i,k}$ and $MTJ_{j,k}$, namely $I_{\rm i,k} + I_{\rm j,k}$. With different reference sensing current, the sense amplifier will have different outputs under given input patterns (corresponds to the high/low resistive state of the MTJs), then different logic functions of the enabled word line can be implemented.

3 Reformulation of Triangle Counting

In this section, we seek to perform TC with massive bitwise operations, which is the enabling technology for in-memory TC accelerator.

3.1 Triangle Counting With Bitwise Operations

Let $\mathbf{A}$ be the adjacency matrix representation of an undirected graph $G(V, E)$, where $\mathbf{A}[i][j] \in \{0, 1\}$ indicates whether there is an edge between vertices i and j. If we compute $\mathbf{A}^2 = \mathbf{A} * \mathbf{A}$, then the value of $\mathbf{A}^2[i][j]$ represents the number of distinct paths of length two between vertices i and j.

In the case that there is an edge between vertex i and vertex j ($\mathbf{A}[i][j] \neq 0$), at the same time i can also reach j through a path of length two ($\mathbf{A}^2[i][j] \neq 0$), where the intermediate vertex is k, then vertices i, j, and k form a triangle. As a result, the number of triangles in G is equal to the number of non-zero elements (nnz) in $\mathbf{A} \circ \mathbf{A}^2$ (the symbol '$\circ$' defines element-wise product), namely

$$TC(G) = nnz(\mathbf{A} \circ \mathbf{A}^2). \tag{1}$$

Since $\mathbf{A}[i][j]$ is either zero or one, we have

$$(\mathbf{A} \circ \mathbf{A}^2)[i][j] = \begin{cases} 0, & \text{if } \mathbf{A}[i][j] = 0; \\ \mathbf{A}^2[i][j], & \text{if } \mathbf{A}[i][j] = 1. \end{cases} \tag{2}$$

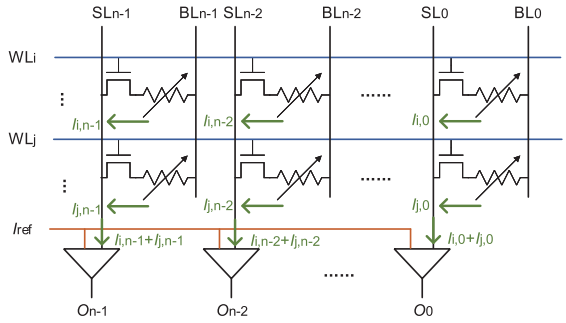


Fig. 2. Computational STT-MRAM array.

According to Equation (2),

$$nnz(\mathbf{A} \circ \mathbf{A}^2) = \sum \sum_{\mathbf{A}[i][j]=1} \mathbf{A}^2[i][j]. \tag{3}$$

Because the element in $\mathbf{A}$ is either zero or one, the bitwise Boolean AND result is equal to that of the mathematical multiplication, thus

$$\begin{aligned} \mathbf{A}^2[i][j] &= \sum_{k=0}^{n} \mathbf{A}[i][k] * \mathbf{A}[k][j] = \sum_{k=0}^{n} AND(\mathbf{A}[i][k], \mathbf{A}[k][j]) \\ &= BitCount(AND(\mathbf{A}[i][*], \mathbf{A}[*][j]^T)), \end{aligned} \tag{4}$$

in which BitCount returns the number of '1's in a vector consisting of '0' and '1', for example, $BitCount(0110) = 2$.

Combining equations (1), (3) and (4), we have

$$\begin{aligned} TC(G) = BitCount(AND(\mathbf{A}[i][*], \mathbf{A}[*][j]^T)), \\ s.t.\ \mathbf{A}[i][j] = 1. \end{aligned} \tag{5}$$

Therefore, TC can be completed by only AND and BitCount operations (massive for large graphs). For each non-zero entry in the adjacency matrix, the corresponding row and column are loaded into STT-MRAM computational memory where each cell consists of one transistor and one MTJ. Consequently, the AND computations are carried out within the STT-MRAM memory, and the bit counter is incremented by the number of 1s in the result of the AND computations. The bit counter will eventually store the total number of triangles in the graph.

3.2 An Illustrative Example

With the reformulated triangle counting method in Section 3.1, for each non-zero element $\mathbf{A}[i][j] = 1$, the ith row ($R_i = \mathbf{A}[i][*]$) and the jth column ($C_j = \mathbf{A}[*][j]^T$) are executed AND operation, then the AND result is sent to a bit counter module for accumulation. Once all the non-zero elements are processed, the value in the accumulated BitCount is the number of triangles in the graph. Fig. 3 demonstrates an illustrative example. The graph has four vertices and five edges, and the adjacency matrix is given. The non-zero elements in the adjacency matrix $\mathbf{A}$ are $\mathbf{A}[0][1]$, $\mathbf{A}[0][2]$, $\mathbf{A}[1][2]$, $\mathbf{A}[1][3]$, and $\mathbf{A}[2][3]$.

1) For $\mathbf{A}[0][1]$, row R_0 = '0110' and column C_1 = '1000' are executed AND operation, then the AND result '0000' is sent to the bit counter and gets a result of zero;

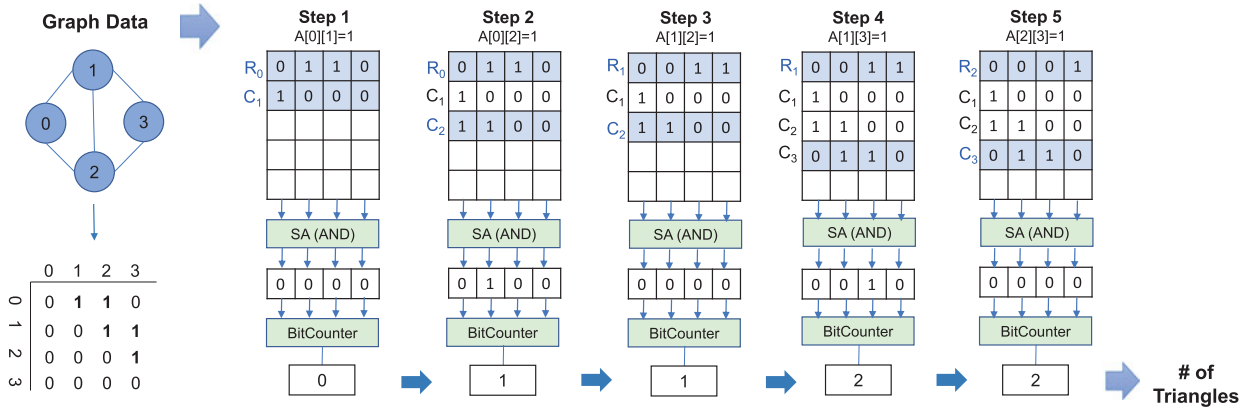


Fig. 3. Demonstrations of triangle counting with AND and BitCount bitwise operations.

2) For $\mathbf{A}[0][2]$, row R_0 = '0110' and column C_2 = '1100' are executed with AND operation and the result is '0100', then the BitCount result of '0100' is one;
3) For $\mathbf{A}[1][2]$, row R_1 = '0011' and column C_2 = '1100' are executed with AND operation, then the AND result '0000' is sent to the bit counter, thus the result remains to be one;
4) For $\mathbf{A}[1][3]$, row R_1 = '0011' and column C_3 = '0110' are executed with AND operation, then the AND result '0010' is executed with BitCount, the bit counter is incremented by one, thus gets a result of two;
5) For $\mathbf{A}[2][3]$, row R_2 = '0001' and column C_3 = '0110' are executed with AND operation, then the AND result '0000' is sent to the bit counter, thus the result remains to be two.

After the process of the last non-zero element $\mathbf{A}[2][3]$ is finished, the accumulated BitCount result is two, as a result, the graph has two triangles (corresponds to triangles $"0-1-2-0"$ and $"1-2-3-1"$ in the graph).

3.3 Discussions on the Reformulated TC

We show that the number of triangles in a given graph can be computed using only AND operations and bit counters on the adjacency matrix of the graph. Once the problem has been framed in this form, the method proposed for triangle counting looks at every non-zero entry in the adjacency matrix and, for each such entry, the corresponding row and column are loaded into STT-MRAM memory where each cell consists of one transistor and one MTJ. AND computations are then carried out within memory and a bit counter is incremented by one if the result of the computations is 1. The bit counter will eventually store the total number of triangles in the graph.

The proposed TC method has the following characteristics: First, it avoids the traditional time-consuming matrix multiplications. Through making the operation data be either zero or one, we can simply implement the original multiplication with Boolean AND logic. Second, the proposed method does not need to store the intermediate results that are larger than one (such as the elements in $\mathbf{A}^2$), which enables high storage efficiency and in-memory computation regularity. Third, it does not need complex control logic. It only needs to iterate the non-zero elements and conduct corresponding AND and BitCount operations.

Given the above three characteristics, and the fact that in-memory computation is suitable for data-intensive applications with relative simple computation and control logic, the proposed reformulated TC method is amenable to highly efficient in-memory computing structure.

4 Sparsity-Aware Graph Data Management for In-Memory Accelerations

Given that the size of the computational memory array is limited, and that most graphs are highly sparse, efficient data flow management is critical for TC accelerations in order to reduce the unnecessary memory and computation requirements. In this part, we will discuss about the data flow management techniques, including the data reuse/replacement and data compression methods, to minimize the needed memory space and computations when being mapped onto the computational memory array.

4.1 Graph Data Reuse and Replacement

The proposed TC method in Section 3 iterates over each non-zero element $\mathbf{A}[i][j]$ in the adjacency matrix $\mathbf{A}$, and loads its corresponding row R_i and column C_j into computational memory for AND operation. As a result, all the non-zero elements in row R_i can reuse this row for computations, and similarly, the non-zero elements in column C_j can reuse this column. We propose data reuse strategy based on this observation.

Without loss of generality, we assume that the non-zero elements are iterated by rows. For each processed row, it needs to be first loaded into the computational memory, then the corresponding columns of the non-zero elements in this row are sequentially loaded for AND computation. In this case, once the computations for all the non-zero elements in a row have been finished, this row will no longer be used in future computations, thus this row can be overwritten by the next to-be-processed row. On the contrary, the corresponding columns might be used again while processing the non-zero elements in other rows. As a result, before loading a certain column into memory for computation, we will first check whether this column has been loaded in previous computations. If it has existed in the computational memory, then it can be reused and save a memory WRITE operation, and if not, the column will be loaded to a spare computational memory space. Overlapping the rows and reusing the

columns can effectively reduce unnecessary space utilization and memory WRITE operations.

Here remains two questions to be answered:

- First, how to decide the row sequence of processing?
- Second, in case that the computational memory is full, by what data replacement policy to swap data?

On selecting the next to-be-processed row, in a greedy way, the local optimal strategy is to choose the next row that has maximum overlaps with the current row on the columns of 1's, and in the ideal case, all the columns should be data hit. However, in case that the size of the matrix is huge then the columns of 1's may not be able to fit in the computational memory. Also, finding the row that overlaps most with the current row will increase non-trivial computational effort. Alternatively, one may do in the zig-zag way: the first row goes from left to right to load the columns with 1, the second row goes from right to left to reuse the columns that are already in. This zig-zag way will work well in case of dense graphs. As for the highly sparse graphs, we will simply process each row sequentially in the order in which the graph data is stored.

Take the case in Fig. 3 as an example, in step 1 and step 2, the two non-zero elements $\mathbf{A}[0][1]$ and $\mathbf{A}[0][2]$ are processed respectively, and corresponding row R_0 and columns C_1 and C_2 are loaded to memory. Next, while processing $\mathbf{A}[1][2]$ and $\mathbf{A}[1][3]$, R_1 will overlap R_0 and reuse existing C_2 in step 3, and load C_3 in step 4. In step 5, to process $\mathbf{A}[2][3]$, R_1 will be overlapped by R_2, and C_3 is reused.

For the data replacement policy, when the computational memory is full and a new column needs to be loaded into the memory for computation, we need to select one candidate column to be swapped out. We know that a good data replacement algorithm should have a low replacement frequency, as a result, data that will not be accessed in the future or will not be accessed for a long time in the future should be swapped out first.

According to our proposed TC method, we need to iterate the non-zero elements in the adjacency matrix by rows. On iterating one certain non-zero element, we need to load the corresponding column for AND and BitCount computations. At the same time, we will also record which columns have been loaded into the computational memory. Therefore, we are able to know about the future computations and the storage status in the computational memory array. Given the above information, when the computational memory is full and a data replacement happens, we are able to locate the column in the computational memory with the longest time between the next visit and swap it out.

Compared to the traditional data replacement strategies such as the LRU (Least Recently Used) policy, which predicts a good choice on choosing to-be-swapped candidate column according to the past computations, our proposed method is able to make the optimal decision with the knowledge of future executions. This can cause the least data replacement frequency, and we name it as Priority data replacement policy.

4.2 Graph Data Compression

To utilize the sparsity of the graph to reduce the memory requirement and unnecessary computation, we propose a data slicing strategy for graph data compression.

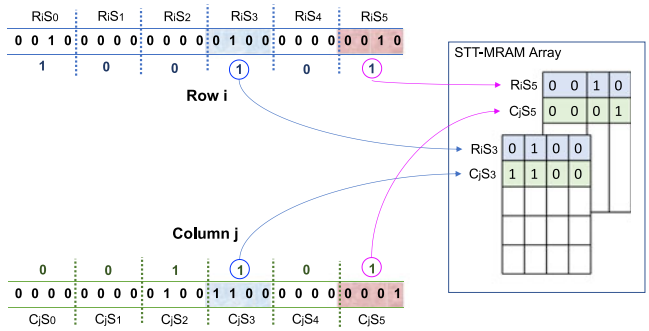


Fig. 4. Sparsity-aware data slicing and mapping.

Assume R_i is the ith row, and C_j is the jth column of the adjacency matrix $\mathbf{A}$ of graph $G(V, E)$. Let the slice length be $|S|$ (namely each slice contains $|S|$ bits), then each row and column has $\lceil \frac{|V|}{|S|} \rceil$ slices. Accordingly, the kth slice in row R_i, which is represented as R_iS_k, can be formulated as

$$R_iS_k = \{\mathbf{A}[i][k * |S|], \cdots, \mathbf{A}[i][(k+1) * |S| - 1]\}.$$

We define slice R_iS_k is *valid* if and only if it has at least one non-zero element, namely

$$\exists \mathbf{A}[i][t] \in R_iS_k, \mathbf{A}[i][t] = 1, t \in [k * |S|, (k+1) * |S| - 1].$$

Similar for the the kth slice of column C_j:

$$C_jS_k = \{\mathbf{A}[k * |S|][j], \cdots, \mathbf{A}[(k+1) * |S| - 1][j]\}.$$

Slice C_jS_k is *valid* if and only if

$$\exists \mathbf{A}[t][j] \in C_jS_k, \mathbf{A}[t][j] = 1, t \in [k * |S|, (k+1) * |S| - 1].$$

Recall that for each non-zero element $\mathbf{A}[i][j] = 1$ in the adjacency matrix, we need to compute the AND of its corresponding row R_i and column C_j. With the proposed row and column slicing methods, we will perform the AND operation in the unit of slices, and we only need to process the valid slice pairs. Namely only when both of the row slice R_iS_k and column slice C_jS_k are valid, we will load the valid slice pair (R_iS_k, C_jS_k) into the computational memory array for AND operation.

Fig. 4 demonstrates an example, after row and column slicing, only the valid slice pairs (R_iS_3, C_jS_3) and (R_iS_5, C_jS_5) will be enabled for AND computation. This gives a glance of the fact that this filter process can reduce the needed computation significantly, especially in the large sparse graphs.

Compression Rate Analysis. Assume that the graph has $|V|$ nodes, $|E|$ edges, the slice length is $|S|$, the sparsity of G is defined as

$$\alpha = 1 - \frac{|E|}{|V|^2}.$$

Therefore, α intuitively demonstrates the probability for an element in the adjacency matrix to be zero. Accordingly, the probability for a slice with length of $|S|$ to be invalid (all elements in the slice should be zero) is $\alpha^{|S|}$. Correspondingly, the probability for a slice to be valid (at least one element in the slice should be non-zero) is $1 - \alpha^{|S|}$. The number of valid slices N_{VS} can be formulated as:

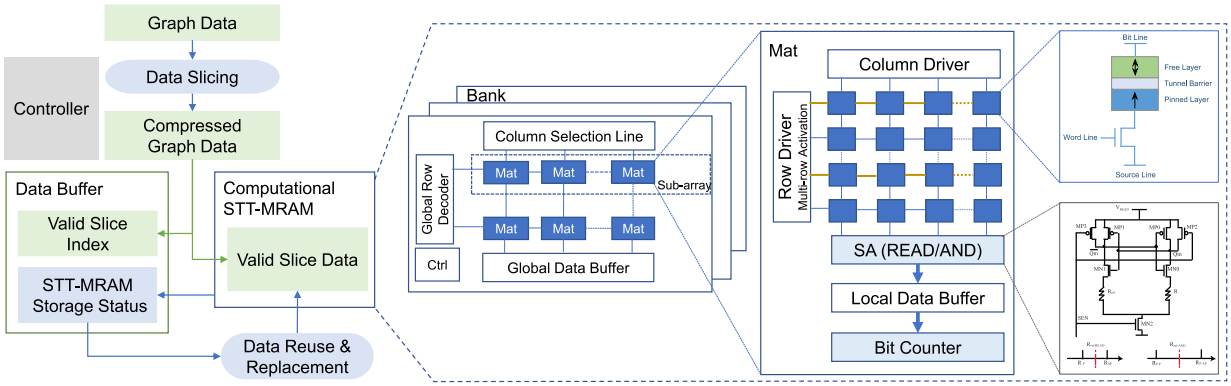


Fig. 5. Overall processing-in-memory architecture.

$$N_{VS} = (1 - \alpha^{|S|}) \cdot \frac{|V|^2}{|S|}.$$

For data compression, we need to store the index of valid slices and the detailed data information of these slices. Assume that we need $|D|$ bits to store the index of slice ($|D| \geq log_2\frac{|V|}{|S|}$), then the overall needed space (in Bytes) for compressed graph G is

$$\begin{aligned}\text{Compressed Graph Size} &= N_{VS} \times \left(\frac{|D| + |S|}{8}\right) \\ &= (1 - \alpha^{|S|}) \cdot \frac{|V|^2}{|S|} \cdot \left(\frac{|D| + |S|}{8}\right).\end{aligned}$$

Without data slicing and compression, the needed storage space (in Bytes) is

$$\text{Ordinary Graph Size} = \frac{|V|^2}{8}.$$

Consequently, the compression rate of the graph data can be expressed as:

$$\begin{aligned}\text{Compression Rate } CR &= \frac{\text{Compressed Graph Size}}{\text{Ordinary Graph Size}} \\ &= \left(1 + \frac{|D|}{|S|}\right) \cdot (1 - \alpha^{|S|})\end{aligned}$$

Therefore, the graph compression rate is determined by the sparsity of the graph, the slice length and the graph size. Fig. 6a demonstrates the compression rate with different graph sparsity and slice length when we use an integer ($|D| = 32$) to store each valid slice index, and Fig. 6b zooms in the figure when the sparsity $\alpha \in (0.9, 1)$. We can see that the graph compression rate is dominated by the graph sparsity, when the sparsity is larger than 0.99, the compression method is expected to have a high compression efficiency. Given that most graphs are highly sparse, the needed space to store the graph can be trivial and the experimental section will demonstrate some results.

More importantly, the proposed format of compressed graph data is friendly for directly mapping onto the computational memory arrays to perform in-memory logic computation. This is because the proposed compression method does not compress the valid slice data, thus does not need complex decompression process.

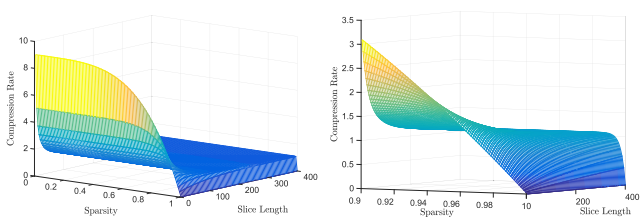
5 Overall Architecture and Implementation

5.1 Overall Architecture Design

Fig. 5 demonstrates the overall architecture of the proposed TC accelerator. First, the graph data will be sliced and compressed, and represented by the valid slice index and corresponding slice data. Consequently, according to the valid slice indexes in the data buffer, the corresponding valid slice pairs are loaded into computational STT-MRAM array for bitwise computation. The storage status of STT-MRAM array (such as which slices have been loaded) is also recorded in the data buffer and utilized for data reuse and replacement.

As for the computational memory array organization, each chip consists of multiple Banks and works as computational array. Each Bank is comprised of multiple computational sub-arrays, which are connected to a global row decoder and a shared global row buffer. Read circuit and write driver of the memory array are modified for processing bitwise logic functions. Specifically, the operation data are stored in different rows in memory arrays. The rows associated with operation data will be activated simultaneously for computing. Sense amplifiers are enhanced with AND reference circuits to realize either READ or AND operations.

Note that in traditional Von-Neumann computing architecture, CPU is the central unit for control and computations, which can efficiently deal with complex computing and control task. In contrast, for in-memory processing, the decentralized processing cores can provide ultra-high parallelism, while they are more suitable for relatively single types of calculations with less control logic, such as the neural network computations. Data-intensive applications (such as the



(a) $\alpha \in (0, 1), |D| = 32$ (b) $\alpha \in (0.9, 1), |D| = 32$

Fig. 6. Compression rate with different sparsity and slice/index length.

triangle counting graph algorithm demonstrated in this paper) which can be reformulated as simple logic computations are amenable to the proposed architecture.

Some nice work on graph computing accelerations has been proposed, such as GraphR [10] and GraphSAR [12]. They point out that large-scale graph calculation problems can be simulated in memristor array in the form of matrix vector operations. However, for graph computing, the implementation of vector matrix multiplication through analog operations faces the problem of accuracy and the additional overhead caused by digital-to-analog/analog-to-digital conversion. The proposed approach in this article reformulates the graph computing problem into basic Boolean logic functions, which can be implemented efficiently in-memory.

5.2 Pseudo-Codes for In-Memory TC Acceleration

Algorithm 1 demonstrates the pseudo-code for TC accelerations with the proposed architecture. It iterates over each edge of the graph (corresponds to each non-zero element in the adjacency matrix) and partitions the corresponding rows and columns into slices, then loads the valid slice pairs onto computational memory for `AND` and `BitCount` computation. In case that there is no enough memory space, it will select one slice with the longest time between the next visit to be swapped out by the new slice. Then repeat the above process until all the non-zero elements in the adjacency matrix are processed, and the accumulated `BitCount` result will be the number of triangles in the graph.

Algorithm 1. Triangle Counting With Processing-In-Memory Architecture

Input: Graph $G(V, E)$.
Output: The number of triangles in G.
$TC\_G = 0$;
Represent G with adjacent matrix **A**;
Iterate the non-zero elements in **A** by rows;
for *each non-zero element* $\mathbf{A}[i][j] = 1$ **do**
  Partition R_i into slices;
  Partition C_j into slices;
  for *each valid slice pair* (R_iS_k, C_jS_k) **do**
    TC_G += **COMPUTE** (R_iS_k, C_jS_k);
return TC_G as the number of triangles in G.
———————————
COMPUTE ($RowSlice$, $ColumnSlice$)
Load $RowSlice$ into memory;
if *ColumnSlice does not exist in the computational memory* **then**
  if *there is no enough space* **then**
    one slice with the longest time between the next visit to be swapped out;
  Load $ColumnSlice$ into memory;
return `BitCount`(`AND`($RowSlice, ColumnSlice$)).

6 Experimental Results

6.1 Experimental Setup

To validate the effectiveness of the proposed methods, comprehensive device-to-architecture evaluations along with two in-house simulators are developed.

TABLE 1
Key Parameters for MTJ Simulations

Parameter	Value
MTJ Surface Length	$40\ nm$
MTJ Surface Width	$40\ nm$
Spin Hall Angle	0.3
Resistance-Area Product of MTJ	$10^{-12}\ \Omega \cdot m^2$
Oxide Barrier Thickness	$0.82\ nm$
TMR	100%
Saturation Field	$10^6\ A/m$
Gilbert Damping Constant	0.03
Perpendicular Magnetic Anisotropy	$4.5 \times 10^5\ A/m$
Temperature	$300\ K$

At the device level, we jointly use the Brinkman model and Landau-Lifshitz-Gilbert (LLG) equation to characterize MTJ [26]. The key parameters for MTJ simulation are demonstrated in Table 1. For the circuit-level simulation, we design a Verilog-A model for 1T1R STT-MRAM device, and characterize the circuit with 45nm FreePDK CMOS library. We design a bit counter module based on Verilog HDL to obtain the number of non-zero elements in a vector. Specifically, we split the vector and feed each 8-bit sub-vector into an 8-256 look-up-table to get its non-zero element number, then sum up the non-zero numbers in all sub-vectors. We synthesis the module with Synopsis Tool and conduct post-synthesis simulation based on 45nm FreePDK. The modified sense amplifier part (to support logic computations) is also simulated in Cadence tool on 45nm FreePDK. After getting the circuit-level simulation results, we integrate the parameters into the open-source NVSim simulator [27] and obtain the memory array performance with wordwidth of 64 bits, 8-way cache configuration. In addition, we develop a simulator in Java for the processing-in-memory architecture, which simulates the proposed function mapping, data slicing and data mapping strategies. Finally, a behavioral-level simulator is developed in Java, taking architectural-level results and memory array performance to calculate the latency and energy that is spent on TC in-memory accelerator.

To provide a solid comparison with other accelerators, we select from the real-life graphs from SNAP dataset [28] (see Table 2), and run comparative baseline intersect-based algorithm on Inspur blade system with the Spark GraphX framework on Intel E5430 single-core CPU. For fair comparisons, our TC in-memory acceleration algorithm also runs on single-core CPU.

6.2 Evaluations of Data Slicing and Compression

For the convenience and efficiency of computing, we can set the slice length to be the multiple of the computer word length. We assume the computer word length to be 64 bits in this paper. Fig. 7 demonstrates the normalized valid slice number when the slice length is 64, 128, and 256, respectively. We can see that the number of valid slices only demonstrate a trivial reduction (on average less than 10%) when the slice length increase from 64 bits to 128/256 bits (each slice has $2\times/4\times$ more bits). Therefore, we set $|S| = 64$ in the following experiments.

Table 3 demonstrates the sparsity of the each benchmark in the SNAP graph dataset and the corresponding

TABLE 2
SNAP Graph Dataset

Graph	# Vertices	# Edges	# Triangles	Description
ego-facebook	4039	88234	1612010	Social circles from Facebook (anonymized)
email-enron	36692	183831	727044	Email communication network from Enron
com-Amazon	334863	925872	667129	Amazon product network
com-DBLP	317080	1049866	2224385	DBLP collaboration network
com-Youtube	1134890	2987624	3056386	Youtube online social network
roadNet-PA	1088092	1541898	67150	Road network of Pennsylvania
roadNet-TX	1379917	1921660	82869	Road network of Texas
roadNet-CA	1965206	2766607	120676	Road network of California
com-LiveJournal	3997962	34681189	177820130	LiveJournal online social network

compression rate when the slice length is 64 and index length is 32. As shown in the second and third columns of Table 3, the real-world graph are highly sparse, which leads to an extreme low compression rate, which validates the theoretical analysis in Section 4.2. As shown in the fourth column of Table 3, the valid slice pairs occupy a very small percentage among the whole slices, and this also leads to a high computation efficiency. The average sparsity of the five largest graphs is 99.999%, with the average compression rate and average percentage of valid slices be 0.01%, This means the proposed data slicing and compression strategy could significantly reduce the needed memory space and computations by 99.99%.

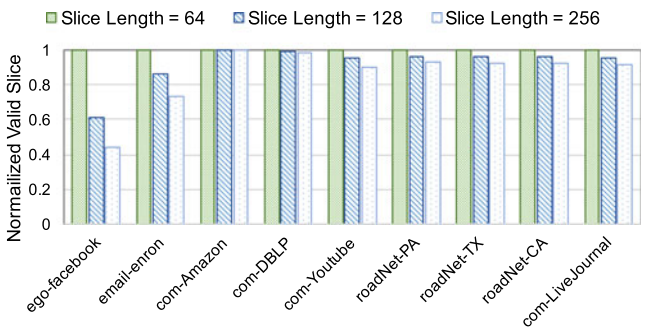


Fig. 7. The number of valid slices with the slice length being 64, 128 and 256, respectively.

TABLE 3
The Sparsity of the Graph Dataset and the Compression Metrics by Data Slicing With Slice Length $|S| = 64$ and Index Length $|D| = 32$

Graph	α*	CR**	$VSR^{\dagger}$
ego-facebook	99.45914%	11.154%	7.017%
email-enron	99.98635%	0.584%	1.483%
com-Amazon	99.99917%	0.078%	0.014%
com-DBLP	99.99896%	0.080%	0.036%
com-Youtube	99.99977%	0.014%	0.013%
roadNet-PA	99.99987%	0.009%	0.013%
roadNet-TX	99.99990%	0.007%	0.010%
roadNet-CA	99.99993%	0.005%	0.007%
com-LiveJournal	99.99978%	0.013%	0.006%

*Sparsity of the graph.
**Compression rate.
†Valid slice pair ratio.

6.3 Evaluations of Data Reuse and Replacement

We know that the first time a data slice is loaded, it is always a miss, and a data hit implies that the slice data has already been loaded and a data reuse has happened. And when the required computational memory is larger than the STT-MRAM computational memory size, at the same time a data miss occurs, then data replacement will happen.

With 8 MB STT-MRAM computational memory array, in Fig. 8, we have listed the ratios of data hit and data miss ratios under LRU and Priority data replacement policies. For the Priority data replacement policy, the data hit and data miss ratios are 60.5% and 39.5%, respectively. The data hit rate implies that the proposed data reuse strategy saves on average 60.5% memory `WRITE` operations.

The five largest graphs, including *com-Youtube*, *roadNet-PA*, *roadNet-TX*, *roadNet-CA*, and *com-LiveJournal*, will have to do data replacement. And the experimental result in Fig. 9 demonstrate that with our proposed Priority data replacement policy, compared with the least recently used

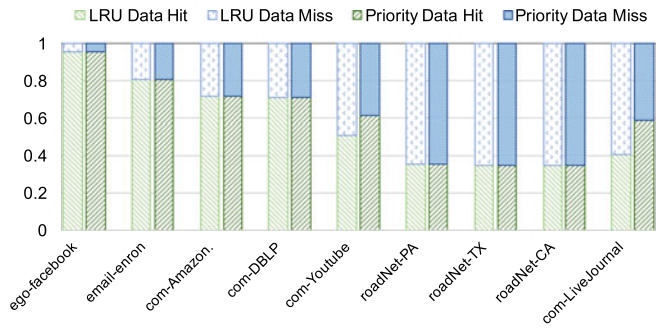


Fig. 8. Data hit and data miss ratios with LRU and priority data replacement strategies.

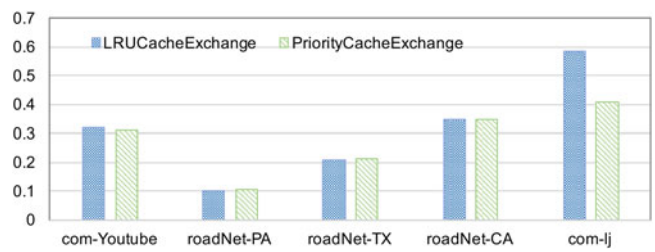


Fig. 9. Data replacement ratio with LRU and priority data replacement strategies.

TABLE 4
Runtime (In Seconds) Comparison Among Our Proposed Methods, CPU, GPU and FPGA Implementations

Dataset	CPU	GPU [3]	FPGA [3]	Proposed Method		
				w/o PIM	TCIM	Priority TCIM
ego-facebook	5.399	0.15	0.093	0.169	0.005	0.005
email-enron	9.545	0.146	0.22	0.8	0.021	0.011
com-Amazon	20.344	N/A	N/A	0.295	0.011	0.011
com-DBLP	20.803	N/A	N/A	0.413	0.027	0.027
com-Youtube	61.309	N/A	N/A	2.442	0.098	0.100
roadNet-PA	77.320	0.169	1.291	0.704	0.043	0.025
roadNet-TX	94.379	0.173	1.586	0.789	0.053	0.030
roadNet-CA	146.858	0.18	2.342	3.561	0.081	0.047
com-LiveJournal	820.616	N/A	N/A	33.034	2.006	1.940
Average					1.0	1.36

(LRU) replacement policy, the number of data replacement is reduced by up to 30.1%.

6.4 Performance and Energy Results

Table 4 compares the performance of our proposed in-memory TC accelerator against a CPU baseline implementation, and the existing GPU and FPGA accelerators.

One can see a dramatic reduction of the execution time in the last columns from the previous three columns. Indeed, without PIM, we achieved an average $53.7\times$ speedup against the baseline CPU implementation because of data slicing, reuse, and replacement. With PIM, another $25.5\times$ acceleration is obtained. Compared with the GPU and FPGA accelerators, the improvement is $9\times$ and $23.4\times$, respectively. It is important to mention that we achieve this with a single-core CPU and 16 MB STT-MRAM computational array. With the optimized Priority data replacement policy (named as Priority TCIM), we can get another $1.36\times$ speedups.

As for the energy savings, as shown in Fig. 10, our approach has $34\times$ less energy consumption compared to the energy-efficient FPGA implementation [3], which benefits from the non-volatile property of STT-MRAM and the in-situ computation capability.

7 Conclusion

In this paper, we propose a new triangle counting (TC) method, which uses massive bitwise logic computation, making it amenable for in-memory implementations. We further propose a sparsity-aware processing-in-memory architecture for efficient in-memory TC accelerations. A straightforward data reuse strategy is proposed to save write operations as well as a data slicing technique to exploit sparsity in the benefit of saving even more write operations. By data slicing, the computation could be reduced by 99.99%, meanwhile the compressed graph data can be directly mapped onto STT-MRAM computational memory array for bitwise operations, and the proposed data reuse and replacement strategy reduces 60.5% of the memory `WRITE` operations. Device-level simulations were carried out to obtain MTJ parameters then used in NVSim to estimate memory array performance. This, in turn, is then used by a behavioral-level simulator developed to compute energy and latency metrics. The device-to-architecture co-simulations demonstrate that our in-memory accelerator achieves improvement in terms of speed and energy efficiency by an order of magnitude over traditional GPU/FPGA accelerators.

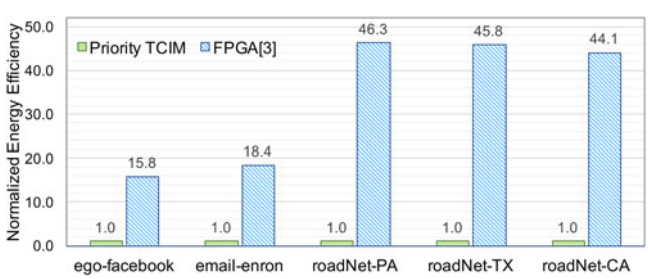


Fig. 10. Normalized results of energy consumption for priority TCIM with respect to FPGA.

References

[1] M. A. Hasan and V. S. Dave, "Triangle counting in large networks: A review," *Wiley Interdisciplinary Reviews: Data Mining Knowl. Discovery*, vol. 8, no. 2, 2018, Art. no. e1226.

[2] V. S. Mailthody *et al.*, "Collaborative (CPU+GPU) algorithms for triangle counting and truss decomposition," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2018, pp. 1–7.

[3] S. Huang *et al.*, "Triangle counting and truss decomposition using FPGA," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2018, pp. 1–7.

[4] V. Seshadri and O. Mutlu, "In-DRAM bulk bitwise execution engine," 2019, *arXiv:1905.09822*.

[5] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proc. 53nd ACM/EDAC/IEEE Des. Automat. Conf.*, 2016, pp. 1–6.

[6] B. Li, B. Yan, and H. Li, "An overview of in-memory processing with emerging non-volatile memory for data-intensive applications," in *Proc. ACM Great Lakes Symp. VLSI*, 2019, pp. 381–386.

[7] S. Angizi, J. Sun, W. Zhang, and D. Fan, "AlignS: A processing-in-memory accelerator for dna short read alignment leveraging sot-mram," in *Proc. 56th ACM/IEEE Des. Automat. Conf.*, 2019, pp. 1–6.

[8] M. Wang *et al.*, "Current-induced magnetization switching in atom-thick tungsten engineered perpendicular magnetic tunnel junctions with large tunnel magnetoresistance," *Nature Commun*, vol. 9, no. 1, 2018, Art. no. 671.

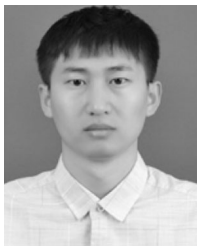
[9] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan, "Computing in memory with spin-transfer torque magnetic RAM," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 3, pp. 470–483, Mar. 2018.

[10] 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.*, 2018, pp. 531–543.

[11] 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.*, 2019, pp. 378–383.
[12] 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 Pacific Des. Automat. Conf.*, 2019, pp. 120–126.
[13] Y. Zhuo *et al.*, "GraphQ: Scalable PIM-based graph processing," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 712–725.
[14] C. Yin *et al.*, "Network science characteristics of brain-derived neuronal cultures deciphered from quantitative phase imaging data," *Scientific Reports*, vol. 10, no. 1, pp. 1–13, 2020.
[15] R. Yang and P. Bogdan, "Controlling the multifractal generating measures of complex networks," *Scientific Reports*, vol. 10, no. 1, pp. 1–13, 2020.
[16] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proc. Int. Symp. Comput. Archit.*, 2015, pp. 336–348.
[17] X. Liu *et al.*, "Reno: A high-efficient reconfigurable neuromorphic computing accelerator design," in *Proc. Des. Automat. Conf.*, 2015, pp. 1–6.
[18] S. G. Ramasubramanian, R. Venkatesan, M. Sharad, K. Roy, and A. Raghunathan, "Spindle: Spintronic deep learning engine for large-scale neuromorphic computing," in *Proc. Int. Symp. Low Power Electron. Des.*, 2014, pp. 15–20.
[19] Z. I. Chowdhury *et al.*, "Efficient in-memory processing using spintronics," *Comput. Archit. Letters*, vol. 17, no. 1, pp. 42–46, 2018.
[20] W. Kang, H. Wang, Z. Wang, Y. Zhang, and W. Zhao, "In-memory processing paradigm for bitwise logic operations in STT–MRAM," *IEEE Trans. Magn.*, vol. 53, no. 11, pp. 1–4, Nov. 2017.
[21] F. Parveen, Z. He, S. Angizi, and D. Fan, "HielM: Highly flexible in-memory computing using STT MRAM," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2018, pp. 361–366.
[22] L. Chang *et al.*, "DASM: Data-streaming-based computing in nonvolatile memory architecture for embedded system," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 27, no. 9, pp. 2046–2059, Sep. 2019.
[23] Y. Zhao *et al.*, "An STT-MRAM based in memory architecture for low power integral computing," *IEEE Trans. Comput.*, vol. 68, no. 4, pp. 617–623, Apr. 2019.
[24] Z. Guo *et al.*, "Spintronics for energy- efficient computing: An overview and outlook," in *Proc. IEEE*, vol. 109, no. 8, pp. 1398–1417, Aug. 2021.
[25] 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. 2018.
[26] J. Yang *et al.*, "Radiation-induced soft error analysis of STT-MRAM: A device to circuit approach," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 3, pp. 380–393, Mar. 2015.
[27] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 7, pp. 994–1007, Jul. 2012.
[28] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," Jun. 2014. [Online]. Available: http://snap.stanford.edu/data

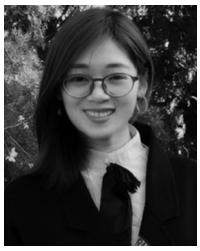
Xueyan Wang (Member, IEEE) received the BS degree in computer science and technology from Shandong University, Jinan, China, in 2013, and the PhD degree in computer science and technology from Tsinghua University, Beijing, China, in 2018. From 2015 to 2016, she was a visiting scholar in the University of Maryland, College Park, MD. She is currently an assistant professor with the School of Integrated Circuit Science and Engineering, Beihang University, Beijing, China. Her research interests include processing-in-memory architectures, AI chip, and hardware security.

Jianlei Yang (Senior Member, IEEE) received the BS degree in microelectronics from Xidian University, Xi'an, China, in 2009, and the PhD degree in computer science and technology from Tsinghua University, Beijing, China, in 2014. He is currently an associate professor in Beihang University, Beijing, China, with the School of Computer Science and Engineering. From 2014 to 2016, he was a postdoctoral researcher with the Department of ECE, University of Pittsburgh, Pittsburgh, Pennsylvania. His current research interests include computer architectures and neuromorphic computing systems. He was the recipient of First/Second place on ACM TAU Power Grid Simulation Contest in 2011/2012. He was a recipient of IEEE ICCD Best Paper Award, in 2013, ACM GLSVLSI Best Paper Nomination, in 2015, IEEE ICESS Best Paper Award, in 2017, ACM SIGKDD Best Student Paper Award, in 2020.

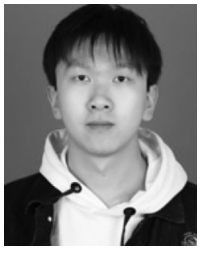


Yinglin Zhao received the MS degree in software engineering from Xidian University, Xi'an, China, in 2017, and currently working toward the PhD degree in electrical engineering in the School of Electronic and Information Engineering, Beihang University, Beijing, China. His research interests include the computer systems architecture and the design of non-volatile memory.

Xiaotao Jia received the BS degree in mathematics from Beijing Jiao Tong University, Beijing, China, in 2011, and the PhD degree in computer science and technology from Tsinghua University, Beijing, China, in 2016. He is currently an associate professor with the School of Microelectronics, Beihang University, Beijing, China. From 2016 to 2019, he was a postdoctoral researcher with the Microelectronics, Beihang University, Beijing, China. His current research interests include spintronic circuits, stochastic computing and Bayesian deep learning.



Rong Yin received the PhD degree from the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China, and the School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China, in 2020. She is currently an associate professor with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. Her current research interests include machine learning, data mining, statistical theory, optimization algorithm, and large-scale kernel methods.



Xuhang Chen received the BS degree in computer science and technology from the Dalian University of Technology, Dalian, China, in 2020, and currently working toward the MS degree in 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.

Gang Qu (Fellow, IEEE) received the BS and MS degrees in mathematics from the University of Science and Technology of China, China, in 1992 and 1994, respectively, and the PhD degree in computer science from the University of California, Los Angeles, Los Angeles, California, in 2000. Upon graduation, he joined the University of Maryland, College Park, Maryland, where he is currently a professor in the Department of Electrical and Computer Engineering and 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 include embedded systems and VLSI CAD with focus on low power system design and hardware related security and trust. He is an associate editor for the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *IEEE Transactions on Emerging Topics in Computing*, *ACM Transactions on Design Automation of Electronic Systems*, *Journal of Hardware and System Security*, *Journal of Computer Science and Technology*, and the *Integration*, *VLSI Journal*. He 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, Hot Picks in Hardware and System Security Workshop, and the IEEE CEDA Hardware Security and Trust Technical Committee.

Weisheng Zhao (Fellow, IEEE) received the PhD degree in physics from the University of Paris Sud, Paris, France, in 2007. He is currently a professor with the School of Integrated Circuit Science and Engineering, Beihang University, Beijing, China. In 2009, he joined the French National Research Center (CNRS), as a tenured research scientist. Since 2014, he has been a distinguished professor with Beihang University, Beijing, China. He has published more than 200 scientifc 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 nano-devices with CMOS circuit and new nonvolatile memory (40-nm technology node and below) like MRAM circuit and architecture design. He is currently the editor-in-chief for the *IEEE Transactions on Circuits and Systems I: Regular Paper*.