Accelerating Graph-Connected Component Computation With Emerging Processing-In-Memory Architecture

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

***Abstract*—Computing the connected component (CC) of a graph is a basic graph computing problem, which has numerous applications like graph partitioning and pattern recognition. Existing methods for computing CC suffer from memory wall problems because of the frequent data transmission between CPU and memory. To overcome this challenge, in this article, we propose to accelerate CC computation with the emerging processing-in-memory (PIM) architecture through an algorithm–architecture co-design manner. The innovation lies in computing CC with bitwise logical operations (such as AND and OR), and the customized data flow management methods to accelerate computation and reduce energy consumption. As a proof of concept, experimental results with computational spin-transfer torque magnetic RAM (STT-MRAM) arrays demonstrate on average 19.8× and 12.4× speedups compared with the CPU and GPU implementations, and a 35.4× energy efficiency improvement over the CPU implementation. Moreover, we investigate the potential associations between graph computing and bitwise Boolean logic, which could help design more general in-memory graph computing accelerators in the future.**

***Index Terms*—Bitwise logical operations, connected component (CC) computation, data flow, processing-in-memory (PIM) architecture.**

Manuscript received 11 October 2021; revised 11 January 2022 and 2 March 2022; accepted 16 March 2022. Date of publication 30 March 2022; date of current version 22 November 2022. The 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 Xiaotao Jia was supported by the Joint Funds of the National Natural Science Foundation of China under Grant U20A20204. The work of Jianlei Yang was supported by the National Natural Science Foundation of China under Grant 62072019. This article was recommended by Associate Editor A. Gamatie. *(Corresponding authors: Xueyan Wang; Xiaotao Jia; Weisheng Zhao.)*

Xuhang Chen, Xueyan Wang, 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@buaa.edu.cn; weisheng.zhao@buaa.edu.cn).

Xiaotao Jia is with the MIIT Key Laboratory of Spintronics, School of Integrated Circuit Science and Engineering, Beihang University, Beijing 100191, China, and also with the Beihang Hangzhou Innovation Institute Yuhang, Beihang University, Hangzhou 310023, China (e-mail: jiaxt@buaa.edu.cn).

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.

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

Digital Object Identifier 10.1109/TCAD.2022.3163628

I. Introduction

CONNECTIVITY of a graph is an essential property in graph theory, which is widely used in pattern recognition [1], graph partitioning [2], graph compression [3], and many other fields [4], [5]. Connected component (CC) computation is one key step in computing the connectivity of the graph. CC computation is not difficult, but it is a data-intensive task, as a result, the frequent data transfers have occupied most of the time and energy. To overcome the challenge, many accelerating methods for CC computation have been proposed, ranging from single-machine algorithms, distributed-memory algorithms, to MapReduce algorithms [6]. Nevertheless, these algorithms are usually based on the Von-Neumann architecture [6], [7], which separates CPU and memory [8]. As a result, the bottleneck caused by the data transfers is only alleviated while not solved [9], [10].

The processing-in-memory (PIM) architecture has been proposed to solve the Von-Neumann bottleneck [11]–[14]. It embeds computing into memory and completes computing while reading data, which reduces a large number of memory accesses and data transmission between CPU and memory. The emerging nonvolatile spin-transfer torque magnetic RAM (STT-MRAM) memory technology could be well combined with PIM. It provides fast read/write speed, low energy consumption, and high endurance among many other advantages [15]–[17], and the characteristic of resistance information storage enables the efficient realization of bitwise logical operations through the accumulation of the electric current [11], [18], [19].

Existing CC computation methods in the literature cannot be directly implemented in memory [6]. On one hand, the existing algorithms for CC computation, including the computation based on union-find disjoint sets and the computation based on depth-first search have many recursive operations, which are difficult to be implemented in memory [20], [21]. On the other hand, graph data compression is one key step for highly efficient graph computing in sparse graphs. Although there have been many data compression methods, such as compressed sparse columns (CSCs), compressed sparse rows (CSRs), and coordinate (COO) [22], they cannot be directly mapped into the computational memory array and need complex decompression processes for computation. To overcome the above

challenges, in this article, we propose a new CC computation method based on basic bitwise logical operations and customized graph compression methods to facilitate efficient in-memory accelerations. The contributions of this article can be summarized as follows.

1) A new algorithm to compute CC with basic bitwise logical operations to facilitate in-memory acceleration.
2) Customized data compression methods by data slicing to speed up computation and save memory, together with an overall PIM architecture.
3) We investigate the internal associations between graph computing and in-memory computing for future general graph computing accelerations.

The remainder of this article is organized as follows. Section II introduces some preliminary knowledge about the CC of the graph, related work, and in-memory computing with STT-MRAM. Section III introduces the proposed CC computation method with bitwise logical operations. Section IV elaborates the sparsity-aware architecture of CC computation in memory. Section V investigates the associations between graph computing and in-memory computing. Section VI demonstrates the experimental results and Section VII concludes this article.

II. Preliminary

A. Graph-Connected Component

In an undirected graph G, if there is a path between vertex V_i and vertices V_j, V_i, and V_j are connected. In a subgraph of G, if any two vertices are connected, it is called the CC of G. Among all CCs, the CC with the largest number of vertices is called the largest CC of G.

Two typical CC computation methods are based on union-find disjoint sets and depth-first search, respectively. The algorithm based on union-find disjoint sets establishes a set for each vertex and finds the parent vertex of each vertex. If two vertices have the same parent vertex, the two sets are merged. Therefore, each set left is a CC of graph G. Finally, all the sets are sorted according to the number of vertices, and the one with the largest number of vertices is the largest CC of the graph G [20]. For the algorithm based on depth-first search, it starts from the first vertex and traverses the neighbor vertices of the vertex according to the adjacency list. The first vertex and all its neighbors are pushed into the stack. Then, the vertex at the top of the stack is popped, and all its neighbors are searched and pushed into the stack. The above process is repeated until the stack is empty, and these vertices that have been pushed into the stack form a CC. There is a CC for each depth-first search. Finally, among all the CCs, the CC with the largest number of vertices is the largest CC of the graph G [21]. We can see that they both involve complex data structures or operations, which are resource consuming to be implemented in memory.

B. Related Work

Due to the wide applications of graph computing and its challenges, accelerating graph computing has been widely studied in the literature. Zhou *et al.* [23] proposed to leverage large external memory for storing massive graph data and FPGA for graph computing acceleration. GraVF [24] is a high-level distribute graph processing framework on FPGAs, which accelerates graph computing through leveraging the vertex-centric paradigm. Wang *et al.* [25] proposed a heterogeneous processing approach for priority scheduling based on a shared-memory CPU-FPGA platform. These works show promising results of accelerating graph computing, however, they are still restricted by the frequent data transfer overhead in Von-Neumann architecture. According to Graphicionado [26], the energy consumption of the storage system in the graph calculation process accounts for more than 90% of the overall system, and one memory access for 32-bit data consumes two orders of magnitude higher energy than mathematical operations [27]. For the PIM-based acceleration approach, most of the existing methods focus on the architecture and circuit-level design, which lacks the algorithm–architecture co-optimization. TCIM [16] proposes to perform triangle counting with bitwise logical operations to enable in-memory implementations, through a hardware–software co-design manner. However, it focuses on the triangle counting problem, and the dataflow management is less effective to directly handle the adjacency matrix. Also, the potential associations between graph computing and bitwise Boolean logic need to be investigated for general in-memory graph computing accelerators.

C. In-Memory Computing With STT-MRAM

Traditional volatile memory technology, such as SRAM and DRAM, faces problems, such as high-power consumption and high-design costs in implementing PIM. New emerging non-volatile memory (such as RRAM, PCM, and MRAM) utilizes the resistance information to store data information, therefore, through the accumulation of current, logic operations can be naturally realized. In the literature, PCM faces issues of high write power, RRAM has been extensively explored and used to implement matrix-vector multiplication for neural network accelerations, with the multibit storage property [28]. Comparatively, STT-MRAM has higher write endurance, while it only has a limited resistance difference between the distinct resistance states of magnetic tunnel junction (MTJ) [29]. Also, 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 exploited to implement bitwise Boolean operations for the general-purpose in-memory computing paradigm, which is adopted in this article.

As shown in Fig. 1, a typical STT-MRAM bit cell consists of an access transistor and an MTJ. It is controlled by bit line (BL), word line (WL), and source line (SL). The relative magnetic orientation of the free and pinned layers determines the resistance offered by the MTJ. The resistance for the parallel configuration R_P corresponds to low resistance and the antiparallel resistance R_{AP} corresponds to high resistance. The `READ` operations are performed by enabling WL and applying a bias voltage (V_{read}) between BL and SL. Through comparing the resultant current flowing through the MTJ (I_P or I_{AP})

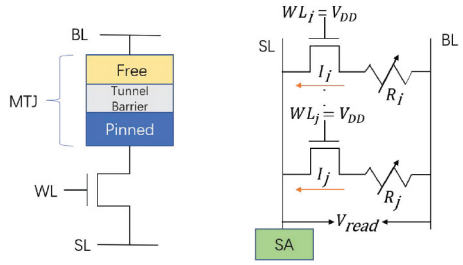


Fig. 1. Typical STT-MRAM bit cell and in-memory computing paradigm with STT-MRAM.

with a global reference to read out the data stored in MTJ. The WRITE operations could be done by enabling WL and applying an appropriate voltage (V_{write}) across BL and SL to pass a current greater than the critical switching current of the MTJ. The logic value written is dependent on the direction of the write current. Bitwise logic operations could be demonstrated in the right part of Fig. 1. It realizes bitwise logical operations by enabling WL_i and WL_j, then a bias voltage V_{read} is applied across BL and SL, the resultant current which feeds into the sense amplifier (SA) is a summation of the currents flowing through each MTJ. Through comparing with the reference sensing currents, various bitwise logical operations of enabled WL could be implemented.

III. Connected Component Computation With Bitwise Logical Operations

In this section, we seek to compute the CC of a graph G with massive bitwise logical operations to make it amenable to in-memory implementation.

A. Reformulation of CC

Let $\mathbf{A}$ be the adjacency matrix of $G(V, E)$, where $\mathbf{A}[i][j] \in \{0, 1\}$ represents whether there is an edge between V_i and V_j. If $\mathbf{A}[i][j] = 1$, it demonstrates that V_j is the neighbor vertex of V_i, thus V_i and V_j belong to the same CC. In fact, if one CC includes V_i, then it should include the vertices that V_i can reach. And the vertices that V_i can reach include the neighbor vertices of V_i and the vertices that the neighbor vertices of V_i can reach. The nonzero elements in the ith row R_i represent the neighbor vertices of V_i. As a result, the CC that involves V_i can be obtained by conducting OR operations among the rows that are associated by vertices' neighborships. The proposed CC computation method is based on this basic observation.

Specifically, we can follow these steps to compute the CC. We use a tag sequence (TS) to indicate which vertices have been processed, and a result sequence (RS) to demonstrate which vertices belong to the same CC.

1) Initialize TS as 1 and RS as 0.
2) Find the first unmarked vertex V_i ($TS[i] = 1$) in the tag sequence and mark it as 1 in the result sequence.
3) Traverse tag sequence and result sequence at the same time to find vertex V_j which satisfies $AND(TS[j], RS[j]) = 1$.
4) Execute $OR(R_j, RS)$, then write the results to the result sequence, and mark V_j in the tag sequence ($TS[j]$) as 0.
5) Repeat steps 3) and 4) until no such V_j exists.
6) Calculate the number of vertices in a CC that involves V_i through a bitcounter.
7) If the number of vertices in the CC is not equal to the total number of vertices and there exists $TS[i] = 1$, repeat steps 2)–6) until all vertices in the tag sequence are marked.
8) The CC with the largest number of vertices is the largest CC.

We can see that the CC could be computed through bitwise logical operations, including AND, OR, and *BitCount*, and these operations can be implemented in-memory conveniently. To sum up, our proposed method has the following advantages. First, it does not need numerous arithmetic operations. The major operations can be converted to basic bitwise logical operations, such as AND and OR. Second, it avoids using complex data structures, such as queue or stack, it only uses a tag sequence to record which vertices have been processed and a result sequence to record which vertices belong to the same CC. Third, this method does not need abundant recursive operations that cannot be implemented easily in memory. As a result, our proposed method is suitable for being implemented in memory.

B. Illustrative Example

Fig. 2 demonstrates an illustrative example of our proposed CC computation method. As the left part of the figures shows, the graph has six vertices and five edges. The adjacency matrix is on the right side of the graph. First, we find the first unmarked vertex V_0 in the tag sequence and mark it as 1 in the result sequence. Second, we traverse the tag sequence and result sequence at the same time to find V_0 that satisfies $AND(TS[0], RS[0]) = 1$. Third, we mark V_0 as 0 in the tag sequence and execute the OR operation between the result sequence and R_0 to obtain all the neighbor vertices of V_0, these vertices and V_0 belong to the same CC. We continue to traverse the new tag sequence and result sequence to find V_1 that satisfies $AND(TS[1], RS[1]) = 1$. We mark V_1 as 0 in the tag sequence and execute the OR operation between the result sequence and R_1. After that, we can know the vertices that V_0 can reach directly and the vertices that V_0 can reach through V_1 belong to the same CC. We repeat the above operations until no such V_i that satisfies $AND(TS[i], RS[i]) = 1$ can be found, and it indicates that all the vertices in the first CC of the graph have been found. Then, we calculate that the number of vertices in the first CC equals four by bitcounter.

The number of vertices in the first CC is not equal to the total number of vertices. Therefore, we find a new unmarked vertex V_4 in the tag sequence, which indicates there is a new CC that involves V_4 in the graph. We traverse the tag sequence and result sequence at the same time to find V_4 that satisfies $AND(TS[4], RS[4]) = 1$. We mark V_4 as 0 in the tag sequence and execute the OR operation between the result sequence and R_4 to obtain the vertices that V_4 can reach directly. We traverse the tag sequence and result sequence at the same time again to find the vertex V_5 that satisfies $AND(TS[5], RS[5]) = 1$. We mark V_5 in the tag sequence as 0 and execute the OR operation

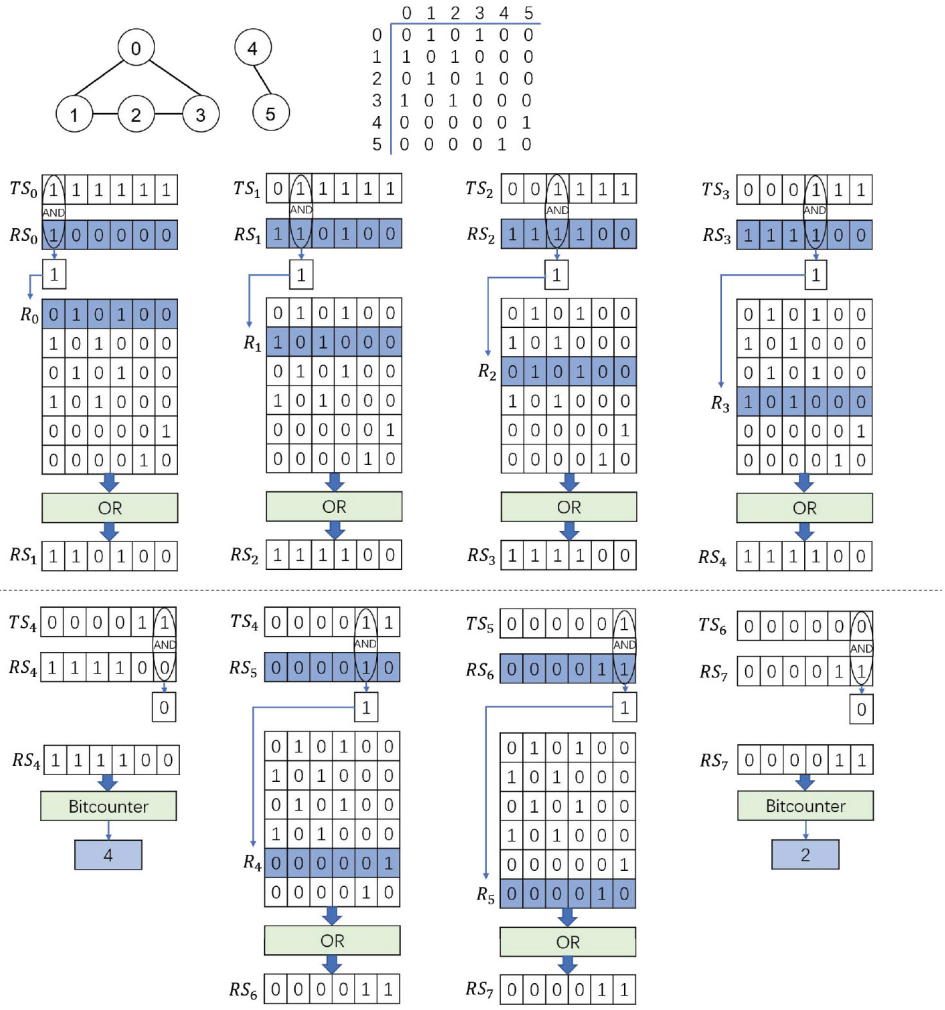


Fig. 2. Demonstrations of CC computation with bitwise logical operations.

between the result sequence and R_5 to obtain the vertices in the new result sequence. These vertices include the vertices that V_4 can reach directly and the vertices that V_4 can reach through V_5. They belong to the same CC. When we traverse the tag sequence and result sequence again, we cannot find such a vertex V_i that satisfies $AND(TS[i], RS[i]) = 1$, and it indicates that all the vertices in the second CC of the graph have been found. After that, we calculate that the number of vertices in the second CC equals two by bitcounter. When we traverse the tag sequence, all the vertices in the tag sequence are marked. It indicates that all the CCs of the graph have been found. Finally, we compare the number of vertices of all the CCs, and the one with the largest number of vertices is the largest CC. The largest CC of the graph has four vertices.

IV. Sparsity-Aware Architecture for In-Memory CC Computation

A. Dataflow Management

To alleviate the bottleneck caused by the frequent data transmission between CPU and memory, we design an algorithm to be amenable to in-memory implementation in the previous section. Next, we will elaborate the data flow management techniques to speed up computation and save memory.

1) Graph Data Slicing: Because most of the graph data is sparse and there are many zero elements in the adjacency matrix which cause unnecessary operations, we design a data slicing strategy for data compression to utilize the sparsity of the graph. Assume R_i is the ith row of the adjacency matrix A of graph $G(V, E)$. We set the length of the slice as $|S|$, which means each slice contains $|S|$ bits data, so each row contains $\lceil |V|/|S| \rceil$ number of slices. The kth slice in R_i, which is represented as R_iS_k, contains data from $k * |S|$th element to $[(k + 1) * |S| - 1]$th element. If there is at least one nonzero element in the slice, we call the slice a valid slice. Exactly, we define that the slice is valid if and only if $\exists A[i][j] \in R_iS_k, A[i][j] = 1, j \in [k * |S|, (k + 1) * |S| - 1]$.

According to the above definition, we know that for each row R_i to be sliced, if V_j is the neighbor vertex of V_i, the slice that contains V_j is valid. And the index of this valid slice is the index of the first element in R_i. Based on this observation, we propose an edge-oriented slicing method. Previously, TCIM [16] proposes a data slicing method based on the adjacency matrix of the graph, which needs to check both the zero and nonzero slices to find the valid ones. In this article,

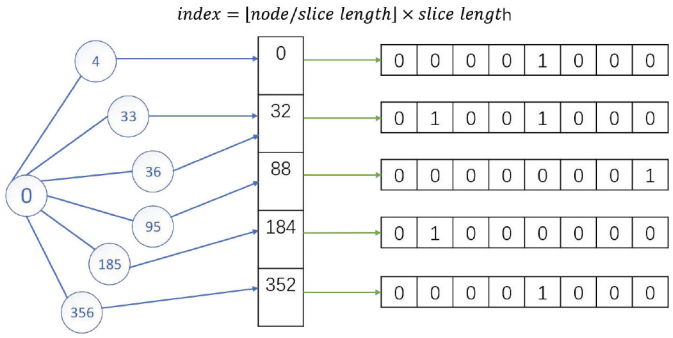


Fig. 3. Edge-oriented data slicing for graph compression.

we propose an edge-oriented slicing method to process the adjacency list. It iterates edges to directly target at the corresponding valid slices, therefore, it can save the expenses of checking the zero slices. Given that any valid slice must correspond to at least one edge, this method can figure out all valid slices with no omissions. Specifically, we slice each row and locate the valid slices by locating its neighbors, which correspond to the edges in the graph. As for the computations, we only need to load the valid slices of R_i into the computational memory array and execute the OR operation between the valid slices and result sequence to obtain the new result sequence.

Fig. 3 demonstrates an example. We slice R_0 of adjacency matrix as an example. All neighbor nodes of V_0 are iterated according to the adjacency list. The neighbor nodes of V_0 are divided into slices according to the slice length. All slices that contain neighbor nodes of V_0 are valid slices. All nodes in each valid slice correspond to an index that is the index of the first element of the valid slice. Assuming that the slice length is 8, the index can be calculated as index $= \lfloor \text{node}/8 \rfloor \times 8$. Meanwhile, we also need to store the data of valid slices for further computation. In this demonstration example, we can see that node V_0 has six neighbors, with the largest one being labeled as 356, the slicing method in TCIM [16] needs to check at least 357 bits in the first row of the adjacency matrix. When the slice length is 8, the proposed data slicing method in this article only needs to deal with $5 \times 8 = 40$ bits (nodes V_{33} and V_{36} correspond to the same slice), which is able to save $(357 - 40)/357 = 88\%$ data processing effort.

2) Result/Tag Sequence Slicing: In our proposed CC computation method, we need to traverse the tag sequence and result sequence at the same time to find the vertex V_i which satisfies $AND(TS[i], RS[i]) = 1$. Assuming that we execute an AND operation in 64-bit units, we need to execute $\lceil |V|/64 \rceil$ numbers of AND operations at most to find a vertex V_i which satisfies $AND(TS[i], RS[i]) = 1$. In order to reduce the needed number of AND operations, we propose to slice tag sequence and result sequence and only process the nonzero slices. Initially, a slice index sequence with the size of $\lceil |V|/|S| \rceil$ is generated. The kth slice in TS and RS, which is represented as TS_k and RS_k, contains data from $k * |S|$th element to $[(k+1) * |S| - 1]$th element. When traversing tag sequence and result sequence, we first traverse the slice index sequence of tag sequence and result sequence to find the slices which satisfy that the corresponding indexes are 1 at the same time, which means that the corresponding slices of tag sequence and result sequence are both valid. Then, we traverse the valid slices of tag sequence and result sequence simultaneously to find the vertex V_i that satisfies $AND(TS[i], RS[i]) = 1$. When updating the tag sequence and result sequence, we need to update the corresponding slice index accordingly.

Fig. 4 demonstrates an example. At the beginning of our proposed method, we find the first unmarked vertex V_i and mark it as 1 in the result sequence. Therefore, there is only one valid slice in the result sequence and all slices of the tag sequence are valid. After executing the OR operation, we traverse the updated part in the tag sequence and result sequence. If there is a "1" element in the slice, the corresponding slice in the result sequence is modified to a valid slice. And if there is no 1 element in the slice of tag sequence, we modify the corresponding slice to be invalid. In this way, when traversing tag sequence and result sequence, we only need to find the valid slices of tag sequence and result sequence according to the slice index, then traverse the valid slices of tag sequence and result sequence at the same time to find the vertex V_i which satisfies $AND(TS[i], RS[i]) = 1$. In the initial execution stage of our method, because the number of valid slices in the result sequence is less, it can obviously save the execution time. This slicing method will greatly reduce the AND operations, but the additional cost is to traverse the updated part of the tag sequence and result sequence to modify the index of slices after each execution.

3) Memory Requirement Analysis: The two slicing strategies proposed in this article require that the slice length is the same. Therefore, it is convenient to determine whether the corresponding slices in the new tag sequence and result sequence are valid after each execution. With our proposed data slicing strategy, we need to store valid slices index and valid slices data. Assuming that the number of valid slices is N_{VS} and the slice length is N, we need to use a four-byte integer to store the index of valid slices. Therefore, the required space for storing the slice index in memory is IndexLength $= N_{VS} \times 4$ bytes. The required space for storing all valid slices data in memory is DataLength $= N_{VS} \times N/8$ bytes. The required space for storing the overall graph G is $N_{VS} \times (N/8 + 4)$ bytes.

For the result/tag sequence slicing strategy, we only need to store the slice index. Assuming that the number of vertices is $|V|$ and the slice length is $|S|$, we also use a four-byte integer to store the index of tag sequence and result sequence. Therefore, the required space in memory for storing the index of tag sequence and result sequence is $4 \times \lceil |V|/|S| \rceil \times 2$ bytes.

B. Overall Architecture and Pseudocodes

1) In-Memory CC Computation Architecture: Fig. 5(a) demonstrates the dataflow management of our proposed CC computation method. The host processor sends a control signal to the computational memory to start PIM computing. And the host processor slices the graph data and compresses the graph with valid slices. Then, the controller sends the valid slice index into the data buffer and loads the valid slices into the computational STT-MRAM array for bitwise logical operations. According to the AND result of the tag sequence and

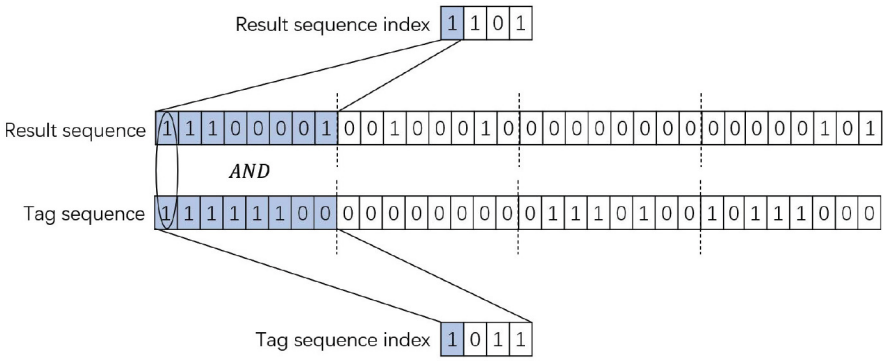


Fig. 4. Tag sequence and result sequence slicing.

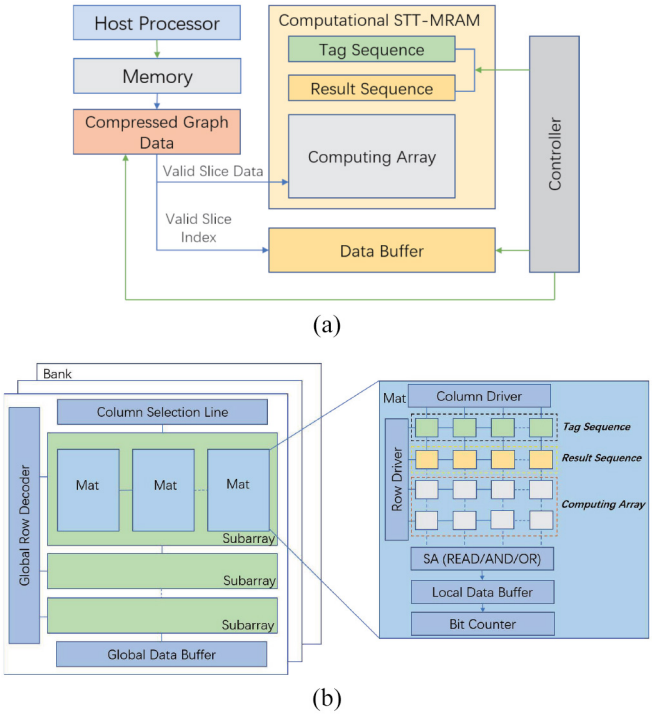


Fig. 5. Overall architecture for CC computation. (a) Dataflow management. (b) Architecture of computational STT-MRAM.

result sequence, the controller selects the corresponding row and enables the valid slices of the selected row and result sequence to execute the OR operation.

As for the computational memory array organization, Fig. 5(b) elaborates the architecture of computational STT-MRAM. Each chip consists of several Banks and works as a computational array. Each Bank consists of several computational memory subarrays, which are connected to a global row decoder and a global data buffer. By modifying the read/write circuits of the memory array, the function of bit-wise logic operation is achieved. Specifically, the operation data are stored in the memory array. The rows with operation data will be activated simultaneously for computing. SAs are enhanced with corresponding circuits to realize the bitwise logical operations.

2) Pseudocodes of In-Memory CC Computation: Algorithm 1 demonstrates the pseudocode for CC computation with the proposed PIM architecture. It iterates every edge of the graph, and slices each row of the adjacency matrix. We just need to load the valid slices of each row R_i into memory and execute the OR operation between valid slices and result sequence. When R_i has been processed, we load other valid slices to exchange valid slices of R_i. In this way, we do not need to load the whole adjacency matrix into memory, we only need to load valid slices into memory, which greatly saves the memory space.

Algorithm 1: CC Computing With the PIM Architecture

Input: Graph $G(V, E)$.
Output: The number of vertices in the largest connected component of G.

```
CC_G = 0;
Represent G with adjacency matrix A;
Tag sequence TS ← 1;
Result sequence RS ← 0;
for each vertex v_i ∈ V with A[i][j] = 1 do
    Partition i-th row R_i into slices;
for each element i in TS do
    if TS[i]==1 then
        CC_G=max(CC_G,COMPUTE (V_i));
    if CC_G == |V| then
        exit;
return CC_G as the number of vertices in the largest connected component.
----------
COMPUTE (V_i)
RS ← 0;
RS[i] ← 1;
while i < |V| do
    if TS[i] ∧ RS[i] == 1 then
        if there is no enough space then
            Replace processed slices with valid slices of R_i;
        else
            Load valid slices of R_i into memory;
        TS[i] ← 0;
        for each valid slice R_iS_K do
            RS_K = OR(R_iS_K, RS_K);
        i ← i + 1;
return BitCount(RS).
```

V. Outreach for Accelerating Graph Computing With In-Memory Computing

As one promising technique to alleviate the Von-Neumann bottleneck, the PIM architecture has attracted more and more attentions from both academia and industry. In this section, we will discuss about the characteristics of graph computing and analyze the inner connections between graph computing and matrix computation, among which matrix computation can be a natural fit for being implemented in memory.

A. Graph Computing Characteristics Analysis

The Von-Neumann bottleneck has been more than severe in the graph computing domain [30]. On one hand, although all vertices of the graph are processed by their storage order in a vertex array, when updating the compute array for each edge after each computation, random accesses will happen. On the other hand, graph algorithms have the nature of a low computation memory rate. It mainly involves some basic operations which only need a short computation time, while it takes a long time to fetch or write the computing results to memory. As a consequence, high bandwidth is required [28].

B. Graph Computing Versus Matrix Computation

A graph could naturally be represented as a matrix: a graph with $|V|$ nodes can be represented with a $(|V| \times |V|)$ matrix, also known as an adjacency matrix, with the intersection of one row and column of the matrix representing the connection relationship of corresponding vertices in the graph [28]. For the PIM architecture, data are stored in the memory in the form of a matrix, at least in the logical point of view. As a result, if the graph computing problem can be transformed into matrix computations, then it can be naturally and efficiently implemented with PIM.

For the edge-related graph computing problem, if we want to explore whether there is an edge between two vertices V_i and V_j, assuming that $\mathbf{A}$ represents the adjacency matrix of the graph, we can know it according to $\mathbf{A}[i][j]$. If $\mathbf{A}[i][j] = 1$, there is an edge between V_i and V_j, and *vice versa*. As a matter of fact, $\mathbf{A}^n[i][j]$ generally represents the number of paths of length n between V_i and V_j. However, matrix multiplication is a complex operation, especially when it is implemented in memory, thus one needs to transform matrix multiplication into logical operations to implement in memory.

For the vertex-related graph computing problem, we usually need to discover the connectivity among different vertices. As for discovering whether two vertices V_i and V_j are connected, one needs to compute the direct path and indirect path between V_i and V_j. The direct path can be known easily from $\mathbf{A}[i][j]$. However, if $\mathbf{A}[i][j] = 0$, we need to compute the indirect path between V_i and V_j. When $\mathbf{A}[i][k] = 1$ and $\mathbf{A}[k][j] = 1$, it means that there is a path between V_i and V_k, at the same time, there is a path between V_k and V_j. Therefore, there is an indirect path $i-k-j$ between V_i and V_j. Therefore, to compute the connectivity between V_i and V_j, we only need to execute $OR(Row_i, Row_k)$. If the jth element in the new sequence that is obtained by the OR operation is 1, it means there is a path between V_i and V_j. Consequently, by converting graph computing into matrix operations and then converting matrix operations into Boolean logical operations, graph computing can be efficiently implemented in the PIM architecture.

Two examples illustrate how graph computing problems are refined for efficient in-memory implementations. The first one is triangle counting. Triangle counting is a classical graph computing problem, which seeks to determine the number of triangles in a graph. In general, a triangle consists of three edges, $\mathbf{A}^3[i][i]$ represents the number of triangles in a graph. However, matrix multiplication is complex to be directly implemented in memory. In the literature, it has been proposed to perform triangle counting with bitwise logical operations to enable in-memory implementations [16]. It proposes to compute $\mathbf{A}^2[i][j]$ that represents the number of paths of length two between V_i and V_j. If $\mathbf{A}[i][j] = 1$, it means that there is an edge between V_i and V_j. The path of length two and the edge between V_i and V_j form a triangle. Because the elements in the matrix are only 0 and 1, the matrix multiplication can be converted to a bitwise Boolean AND result. Therefore, the number of triangles in the graph is equal to $\sum_{\mathbf{A}[i][j]=1} BitCount(AND(Row_i, Column_j))$. The second one is the CC computation proposed in this article. CC computation is also a basic graph computing problem. The proposed method in this article is to compute the CC with bitwise logical operations (such as AND/OR) to enable in-memory acceleration.

In a similar way, many classical graph computing problems could be solved by the combination with such bitwise logical operations in memory. Hopefully, this article could help and motivate more general in-memory graph computing accelerators in the future.

TABLE I
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	$300\ K$

VI. Experimental Results

A. Experimental Setup

To validate the effectiveness of our proposed method, we develop comprehensive device-to-architecture evaluations along with two internal simulators. From the device level, we characterize MTJ from the Landau–Lifshitz–Gilbert (LLG) equation and the Brinkman model.

Table I shows the specific information of MTJ. From the circuit-level simulation level, we design a Verilog-A model for the 1T1R STT-MRAM device and characterize the circuit with a 45 nm FreePDK CMOS library.

TABLE II
SNAP GRAPH DATASET

Graph	Vertices	Edges	Vertices in largest CC	Description
p2p-Gnutella06	8,717	31,525	8,717	Directed Gnutella P2P network from August 6 2002
p2p-Gnutella31	62,586	147,892	62,561	Directed Gnutella P2P network from August 31 2002
email-Enron	36,692	183,831	33,696	Enron email network
email-EuAll	265,214	420,045	224,832	Email network of a large European Research Institution
soc-Slashdot0922	82,168	948,464	82,168	Slashdot Zoo social network
web-NotreDame	325,729	1,497,134	325,729	University of Notre Dame web graph
amazon0302	262,111	1,234,877	262,111	Amazon product co-purchasing network
amazon0505	410,236	3,356,824	410,236	Amazon product co-purchasing network

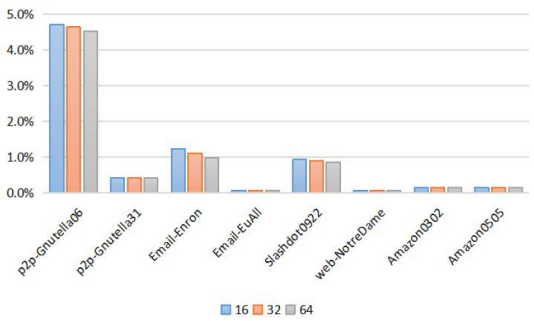


Fig. 6. Percentage of valid slices with the slice length being 16, 32, and 64, respectively.

We also design a bit counter module based on Verilog HDL to acquire the number of nonzero elements in a vector. Specifically, we split the vector and feed each 8-bit subvector into an 8–256 lookup table to obtain the number of nonzero elements, then sum up the nonzero numbers in all subvectors. We synthesis the module with Synopsis Tool and conduct a postsynthesis simulation based on 45 nm FreePDK. After getting the circuit-level simulation results, we integrate the parameters into the open-source NVSim simulator [31] and acquire the memory array performance.

In addition, we use Python to develop a simulator based on MRAM, which simulates the process of computing CC and data slicing. In order to make the experimental results more accurate and convincing, we use the real graph in Stanford large network dataset collection [32] (see Table II) and compare it with the method of computing CCs in the framework of GraphX [33] running on CPU and GPU. The configuration of the adopted CPU is Intel Core i7-7700HQ with 2.8 GHz and eight cores. And the configuration of the adopted GPU is NVIDIA Tesla V100 with 1370 MHz. Our proposed method also runs on the CPU and STT-MRAM computational array which is set to be 16 MB.

B. Benefits of Data Slicing

To utilize the sparsity of the graph data, we propose customized data slicing strategies. From Fig. 6, we can see that the percentage of valid slices in the total slices decreases slowly when the slice length increases from 16 to 64 bits. This shows that the adjacency matrix of the graph is sparse enough, thus the slice length has little effect on the number of valid slices. On the other hand, in general computer, a word is 64 bits. Therefore, we set $|S| = 64$ in the following experiments.

TABLE III
VALID SLICE DATA SIZE (MB) AND PERCENTAGES

Graph	Valid slice data size	Valid slice percentage
p2p-Gnutella06	0.618	4.523%
p2p-Gnutella31	2.819	0.402%
email-Enron	2.343	0.971%
email-EuAll	7.287	0.058%
soc-Slashdot0922	10.179	0.843%
web-NotreDame	10.424	0.055%
amazon0302	16.738	0.136%
amazon0505	41.631	0.138%

TABLE IV
REDUCTION PERCENTAGE OF AND OPERATIONS

Graph	Reduction of AND operations
p2p-Gnutella06	99.235%
p2p-Gnutella31	99.206%
email-Enron	99.186%
email-EuAll	99.180%
soc-Slashdot0922	99.155%
web-NotreDame	99.220%
amazon0302	99.221%
amazon0505	99.221%
average	99.203%

Table III shows the memory space required for all the valid slices in each graph. The largest graph amazon0505 needs 42 MB. Based on our proposed method, we could load valid slices into STT-MRAM computational memory by each row. When valid slices of one row have been processed, exchange valid slices of other rows into STT-MRAM computational memory. In this way, 16 MB is enough to store the valid slices of one row and the required memory space is reduced significantly. The third column in Table III shows the proportion of the number of valid slices in the total number of slices in the adjacency matrix of the graph. It can be seen that the percentages of valid slices in most graphs are less than 1.0%. Therefore, the proposed data slicing method could significantly reduce 99% OR operations compared to the needed OR operations without slicing. As the size of the graph increases, the proportion of the valid slices keeps decreasing. When the size of the graph reaches a certain scale, the proposed data slicing method could significantly reduce 99.9% OR operations.

Table IV shows the reduction percentage of AND operations. It can be seen that the average reduction percentage of AND operations in our dataset is 99.2%. Therefore, due to our proposed result/tag sequence slicing strategy, a large number of AND operations are decreased. Compared to the needed AND operations without slicing, only 0.8% AND operations

TABLE V
RUNTIME (IN SECONDS) COMPARISON AMONG OUR PROPOSED METHODS, CPU, AND GPU IMPLEMENTATIONS

Dataset	CPU	GPU	This Work	
			w/o PIM	w/ PIM
p2p-Gnutella06	0.013	0.010	2.428	0.0006
p2p-Gnutella31	0.108	0.082	25.538	0.0041
email-Enron	0.093	0.044	14.323	0.0028
email-EuAll	0.394	0.246	222.114	0.036
soc-Slashdot0922	0.234	0.123	62.045	0.012
web-NotreDame	0.547	0.316	35.243	0.044
amazon0302	0.656	0.468	243.001	0.038
amazon0505	1.657	0.995	647.733	0.095

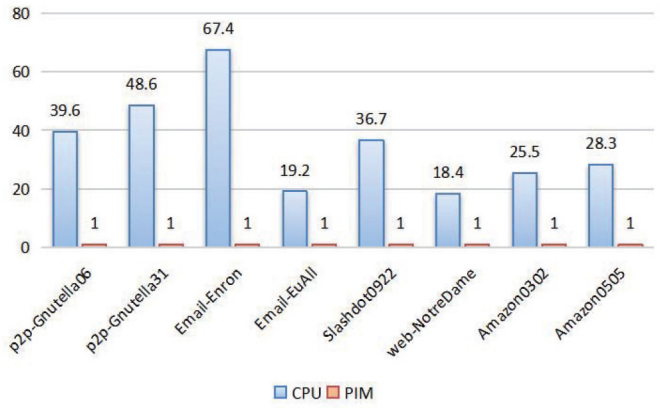


Fig. 7. Normalized results of energy consumption for PIM compared to CPU implementation.

are needed. These experimental results demonstrate that the proposed data slicing can save a large number of computing resources and execution time.

C. Performance and Energy Results

Table V compares the performance of our proposed in-memory CC accelerator with the CPU implementation of our method and the existing framework GraphX for CC computation on CPU and GPU (without data slicing).

In terms of running speed, although our proposed method for CC computation in CPU is slower than the GraphX framework, our proposed method successfully converts complex operations into bitwise logical operations and it is more adapted at computing in memory. With the emerging nonvolatile STT-MRAM memory technology, our proposed in-memory CC accelerator obtains an average 19.8× and 12.4× speedups compared to the existing advanced framework based on CPU and GPU for CC computation, respectively. With the PIM architecture, data transmission between CPU and memory is substantially reduced, and our proposed in-memory CC accelerator achieves an inspiring performance.

Fig. 7 shows the energy savings in our proposed CC accelerator compared with CC computation on CPU framework (considering that GPU is less energy efficient). The geometric mean of energy savings of our CC accelerator compared to the existing framework on CPU is 35.4× due to several properties, such as near zero leakage, high density, and nonvolatility of STT-MRAM, and low-power advantages of bitwise logical operations.

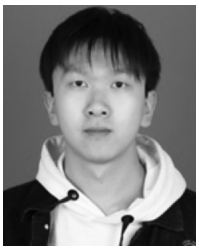
VII. CONCLUSION

In this article, we propose a new method with bitwise logical operations for CC computation, which is suitable for the implementation in memory. Furthermore, we design a PIM architecture for efficiently computing CC in a graph: by data slicing, the OR operations could be reduced by 99.9%, and the AND operations could be reduced by 99.2%. Meanwhile, compressed graph data could be directly mapped onto the STT-MRAM computational memory array for bitwise logical operations. Device-to-architecture co-simulation demonstrates that compared with the CPU and GPU implementations, our proposed CC computation method achieves 19.8× and 12.4× speedups, respectively, and a 35.4× energy efficiency improvement over the CPU implementation.

REFERENCES

[1] L. He, X. Zhao, Y. Yang, H. Tang, and Y. Chao, "Fast connected-component labeling for binary hexagonal images," in *Proc. IEEE Region 10 Conf. TENCON*, 2013, pp. 1–4.
[2] Y. Lim, W.-J. Lee, H.-J. Choi, and U. Kang, "Discovering large subsets with high quality partitions in real world graphs," in *Proc. BigComp*, 2015, pp. 186–193.
[3] Y. Lim, U. Kang, and C. Faloutsos, "Slashburn: Graph compression and mining beyond caveman communities," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 12, pp. 3077–3089, Dec. 2014.
[4] C. Wang, C. Xu, and A. Lisser, "Bandwidth minimization problem," in *Proc. 10ème Conférence Francophone de Modélisation, Optim. et Simul. (MOSIM)* 2014.
[5] M. J. DinnEeN, M. Khosravani, and A. Probert. "Using Opencl for Implementing Simple Parallel Graph Algorithms," 2012. [Online]. Available: hgpu.org
[6] H. M. Park, N. Park, S. H. Myaeng, and U. Kang, "PACC: Large scale connected component computation on hadoop and spark," *PLoS ONE*, vol. 15, no. 3, 2020, Art. no. e0229936.
[7] D. P. Playne and K. Hawick, "A new algorithm for parallel connected-component labelling on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1217–1230, Jun. 2018.
[8] D. Ielmini and H.-S. P. Wong, "In-memory computing with resistive switching devices," *Nat. Electron.*, vol. 1, no. 6, pp. 333–343, 2018.
[9] L. Koskinen, J. Tissari, J. Teittinen, E. Lehtonen, M. Laiho, and J. H. Poikonen, "A performance case-study on memristive computing-in-memory versus von neumann architecture," in *Proc. IEEE DCC*, 2016, p. 613.
[10] C. Cen, K. Li, A. Ouyang, T. Zhuo, and K. Li, "GFlink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data," in *Proc. Int. Conf. Parallel Process.*, 2016, pp. 542–551.
[11] Z. Guo *et al.*, "Spintronics for energy-efficient computing: An overview and outlook," *Proc. IEEE*, vol. 109, no. 8, pp. 1398–1417, Aug. 2021.
[12] J. P. Wang and J. D. Harms, "General structure for computational random access memory (CRAM)," U.S. Patent 9,224,447,B2, 2015.
[13] S. Li *et al.*, "Heterogeneous systems with reconfigurable neuromorphic computing accelerators," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2016, pp. 125–128.
[14] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level PIM offloading in graph computing frameworks," in *Proc. IEEE Int. Symp. High Perform. Comput. Architect. (HPCA)*, 2017, pp. 457–468.
[15] 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.
[16] X. Wang *et al.*, "Tcim: Triangle counting acceleration with processing-in-mram architecture," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, 2020, pp. 1–6.
[17] W. Kang, H. Wang, Z. Wang, Y. Zhang, and W. Zhao, "In-memory processing paradigm for bitwise logic operations in STTŰMRAM," *IEEE Trans. Magn.*, vol. 53, no. 11, pp. 1–4, Nov. 2017.
[18] Y. Pan *et al.*, "An STT-MRAM based reconfigurable computing-in-memory architecture for general purpose computing," *CCF Trans. High Perform. Comput.*, vol. 2, no. 3, pp. 272–281, 2020.

[19] X. Wang, J. Yang, Y. Zhao, X. Jia, G. Qu, and W. Zhao, "Hardware security in spin-based computing-in-memory: Analysis, exploits, and mitigation techniques," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 16, no. 4, pp. 1–18, 2020.
[20] H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *J. Comput. Syst. Sci.*, vol. 30, no. 2, pp. 209–221, 1983.
[21] B. Awerbuch, "A new distributed depth-first-search algorithm," *Inf. Process. Lett.*, vol. 20, no. 3, pp. 147–150, 1985.
[22] L. Song, Y. Zhuo, X. Qian, L. Hai, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *Proc. IEEE Int. Symp. High Perform. Comput. Architect.*, 2018, pp. 531–543.
[23] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and energy-efficient graph processing on FPGA," in *Proc. IEEE 24th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2016, pp. 103–110.
[24] N. Engelhardt and H. K.-H. So, "GraVF: A vertex-centric distributed graph processing framework on FPGAs," in *Proc. 26th Int. Conf. Field Program. Logic Appl. (FPL)*, 2016, pp. 1–4.
[25] Y. Wang, J. C. Hoe, and E. Nurvitadhi, "Processor assisted worklist scheduling for FPGA accelerated graph processing on a shared-memory platform," in *Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2019, pp. 136–144.
[26] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitect. (MICRO)*, 2016, pp. 1–13.
[27] S. Han *et al.*, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Comput. Architect. News*, vol. 44, no. 3, pp. 243–254, 2016.
[28] X. Qian, "Graph processing and machine learning architectures with emerging memory technologies: A survey," *Sci. China Inf. Sci.*, vol. 64, no. 6, pp. 1–25, 2021.
[29] M. Wang *et al.*, "Current-induced magnetization switching in atom-thick tungsten engineered perpendicular magnetic tunnel junctions with large tunnel magnetoresistance," *Nat. Commun.*, vol. 9, no. 1, p. 671, 2018.
[30] 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.
[31] 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.
[32] J. Leskovec and A. Krevl. "SNAP Datasets: Stanford Large Network Dataset Collection," Jun. 2014. [Online]. Available: http://snap.stanford.edu/data
[33] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "{GraphX}: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2014, pp. 599–613.



Xuhang Chen received the B.S. degree in computer science and technology from the 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. Her current research interests include processing-in-memory architectures, AI chip, and hardware security.

Xiaotao Jia (Member, IEEE) received the B.S. degree in mathematics from Beijing Jiaotong 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 of Microelectronics from 2016 to 2019. His current research interests include spintronic circuits, stochastic computing, and Bayesian deep learning.

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.

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 at Los Angeles, Los Angeles, CA, USA, 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 Institute for Systems Research. He is the Director of Maryland Embedded Systems and Hardware Security Lab and Wireless Sensors Laboratory. His primary research interests are in the area of embedded systems and very large-scale integration (VLSI) CAD with focus on low-power system design and hardware related security and trust.

Dr. Qu is serving as an Associate Editor for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, *ACM Transactions on Design Automation of Electronic Systems*, IEEE EMBEDDED SYSTEMS LETTERS, and *Integration, the VLSI Journal*.

Weisheng Zhao (Fellow, IEEE) received the Ph.D. 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), Paris, 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 of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART I: REGULAR PAPERS.