An Efficient ReRAM-based Accelerator for Asynchronous Iterative Graph Processing

JIN ZHAO, Huazhong University of Science and Technology, Wuhan, China
YU ZHANG, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
DONGHAO HE, Huazhong University of Science and Technology, Wuhan, China
QIKUN LI, HUST, Wuhan, China
WEIHANG YIN, Huazhong University of Science and Technology, Wuhan, China
HUI YU, Huazhong University of Science and Technology, Wuhan, China
HAO QI, Huazhong University of Science and Technology, Wuhan, China
XIAOFEI LIAO, Huazhong University of Science and Technology, Wuhan, China
HAI JIN, School of computer science and technology, Huazhong University of Science and Technology, Services Computing Technology and System Laboratory / Cluster and Grid Computing Laboratory, Wu hanwu, China
HAIKUN LIU, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
LINCHEN YU, Huazhong University of Science and Technology, Wuhan, China
ZHANG ZHAN, Zhejiang Lab, Hangzhou, China

Graph processing has become a central concern for many real-world applications and is well-known for its low compute-to-communication ratios and poor data locality. By integrating computing logic into memory, resistive random access memory (ReRAM) tackles the demand for high memory bandwidth in graph processing. Despite the years' research efforts, existing ReRAM-based graph processing approaches still face the challenges of *redundant computation overhead*. It is because the vertices of many subgraphs are ineffectively and repeatedly processed over the ReRAM crossbars for lots of iterations so as to update their states according to the vertices of other subgraphs regardless of the dependencies among the subgraphs. In this paper, we propose *ASGraph*, a dependency-aware ReRAM-based graph processing accelerator that overcomes the aforementioned performance bottlenecks. Specifically, ASGraph dynamically constructs the subgraph based on the dependencies between vertices' states and then detects constructed subgraph that owns high value (it is likely that it has accumulated many state

This is a new article, not an extension of a conference paper.

This paper is supported by National Key Research and Development Program of China (No.2023YFB4502300), Key Research and Development Program of Hubei Province (No. 2023BAB078), Knowledge Innovation Program of Wuhan-Basi Research (No. 2022013301015177), and Huawei Technologies Co., Ltd (No. YBN2021035018A6).

Authors' Contact Information: Jin Zhao, zjin@hust.edu.cn; Yu Zhang, zhyu@hust.edu.cn; Donghao He, hdh@hust.edu.cn; Qikun Li, lqk2021@hust.edu.cn; Weihang Yin, hannyin@hust.edu.cn; Hui Yu, huiy@hust.edu.cn; Hao Qi, theqihao@hust.edu.cn; Xiaofei Liao, xfliao@hust.edu.cn; Hai Jin, hjin@hust.edu.cn; Haikun Liu, hkliu@hust.edu.cn, National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, Hubei, 430074, China; Linchen Yu (corresponding author), linchenyu@hust.edu.cn; School of Cyber Science and Engineering, Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, Hubei, 430074, China; Zhan Zhang, zhanzhang@zhejianglab.com; Zhejiang-HUST Joint Research Center for Graph Processing, Zhejiang Lab, Kechuang Avenue, Hangzhou, Zhejiang, 311121, China.

ACM 1544-3973/2024/11-ART
https://doi.org/10.1145/3689335

propagations from its neighbors and is able to affect more other neighbors) to be preferentially processed. In this way, it makes the vertex states propagate along the dependencies between vertices as much as possible to reduce the redundant computation. Besides, ASGraph employs a hybrid processing scheme to accelerate the state propagations of the tightly connected subgraph, thereby minimizing the redundant computations. Experimental results show that ASGraph achieves 25.5× and 4.8× speedup and 70.8× and 2.2× energy saving on average compared with the state-of-the-art ReRAM-based graph processing accelerators, i.e., GraphR and GaaS-X, respectively.

CCS Concepts: • **Computer systems organization → Architectures**; • **Hardware → Memory and dense storage**.

Additional Key Words and Phrases: processing-in-memory, ReRAM, asynchronous graph processing, convergence speed

1 Introduction

Graph processing is widely used to capture complex relationships between real-world objects and plays a crucial role in various domains, including natural language processing [4, 10, 27], social network analysis [6, 8], machine learning [14, 25, 31, 56] and recommendation systems [23, 40, 46]. Despite the availability of many hardware accelerators [7, 17, 28, 35, 36, 50, 55, 64, 66] designed to enhance high-performance graph processing, they still fall short in effectively addressing the memory bottleneck inherent in graph processing because of their heavy random accesses, low compute-to-communication ratio, and high memory bandwidth requirements. In contrast to these traditional von Neumann architecture-based accelerators, several *resistive random access memory* (ReRAM)-based accelerators [5, 11, 42, 44, 69, 70] are recently designed to alleviate the memory bottleneck of graph processing by harnessing the massive parallel in-situ computation capability offered by ReRAM. To support graph processing, these ReRAM-based solutions typically organize the ReRAM cells in a crossbar architecture to enable efficient graph operations through performing *sparse matrix-vector multiplication* (SpMV) over the ReRAM crossbar, where the graph is typically split into a series of small size matrix-formatted subgraphs to match the crossbar size and alleviate the problem of the processing of the non-existent edges[1]. However, existing ReRAM-based solutions still encounter the following problem.

The new vertex states within subgraphs are irregularly propagated to the vertices in other subgraphs, resulting in significant *redundant computation overhead*. This inefficiency greatly limits the exploitation of the extensive parallelism capabilities inherent in the ReRAM-based architecture. Specifically, existing ReRAM-based solutions [11, 44, 69, 70] typically adopt the subgraph as the basic unit for processing and handle each subgraph at most once in an iteration. Therefore, they usually require lots of iterations to propagate the new vertex states of each subgraph to their successors, which incurs slow convergence speed. Although existing asynchronous graph processing solutions [59, 68] can be applied for the ReRAM-based architectures, the states of the different vertices in each matrix-formatted subgraph are concurrently and irregularly propagated to other matrix-formatted subgraphs, which leads to significant redundant computations. Besides, the subgraphs of the dependency chain[2] may be processed by different ReRAM crossbars concurrently, in which the vertex states of each subgraph may be updated according to the stale vertex states of its neighbors. Thus, many subgraphs need to be frequent reprocessed for existing ReRAM-based solutions to update their vertices' states repeatedly. As a result, it wastes much time for the repeated and redundant processing of the graph data associated with the same subgraph at different times.

After analyzing the characteristics of SpMV-based graph processing, we have made the following observations. The new vertex states of each subgraph can be quickly propagated to the others when the subgraphs are handled along their dependencies as well as the vertices that are tightly connected are assigned to the same subgraphs for processing, because the vertices' states are inherently propagated along the dependencies among these

[1]The weights of these edges in the corresponding adjacency matrix are zero.
[2]Each dependency <A, B> indicates that the vertex states corresponding to the subgraph B are depended on that corresponding to the subgraph A.

vertices and the new vertex states associated with the tightly connected subgraphs can be propagated to many vertices within these subgraphs by iterating them multiple times. Based on the above observations, we develop a dependency-aware ReRAM-based accelerator called *ASGraph*, which can efficiently perform the asynchronous graph processing for better performance.

Different from the existing solutions, ASGraph employs an efficient *dependency-aware asynchronous processing approach* toward ReRAM-based graph processing to eliminate the redundant computations. In detail, ASGraph starts from each active vertex to explore the graph along the data dependencies among the vertices to generate the subgraphs with good internal connectivity. Then, ASGraph detects the explored subgraphs that have accumulated many state propagations from their neighbors and then classifies them as *high-value subgraphs*. After that, these high-value subgraphs are preferentially assigned to be mapped on the same ReRAM crossbar for processing, achieving the regularized state propagations. After that, many subgraphs do not need to be reprocessed once they have been processed, because no new vertex states will be propagated from other subgraphs. Besides, ASGraph proposes a hybrid processing scheme to handle each row of subgraphs by iterating the tightly connected subgraph in this row multiple times and handling other subgraphs in this row only once, minimizing the redundant computations. In this way, the new vertex states within each subgraph are regularly propagated along the dependencies among the vertices' states for the minimal redundant computation overhead.

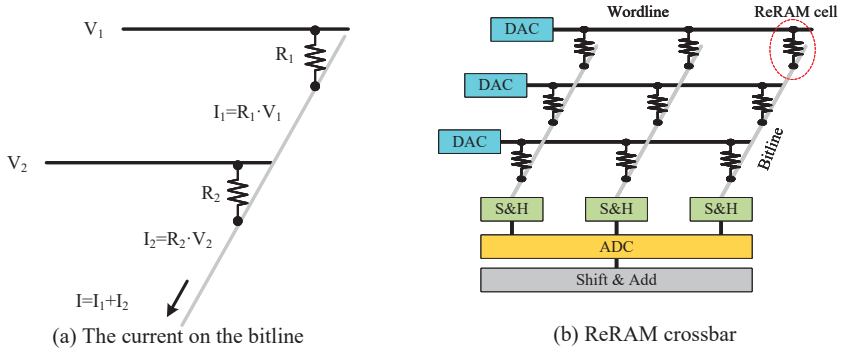
Our main contributions are as follows:

- We reveal the intrinsic causes for the problems of the redundant computation overhead of existing ReRAM-based graph processing solutions.
- We propose a dependency-aware subgraph construction method and an efficient value-driven scheduling mechanism to regularize the state propagations and apply a hybrid processing scheme to minimize the redundant computation overhead.
- We conduct comprehensive experiments to demonstrate the advantages of ASGraph. The results show that ASGraph outperforms the cutting-edge CPU and GPU solutions (i.e., Ligra [43], HotGraph [59], Gunrock [48], and Scaph [68]) by 1514.9×, 661.1×, 39.6×, and 52.8× and achieves 8064.2×, 3347.8×, 521.9×, and 676.5× energy saving on average, respectively. Compared to the state-of-the-art ReRAM-based graph processing accelerators (i.e., GraphR [44] and GaaS-X [5]), ASGraph achieves 25.5× and 4.8× performance improvement and 70.8× and 2.2× energy saving on average, respectively.

2 Background and Motivation

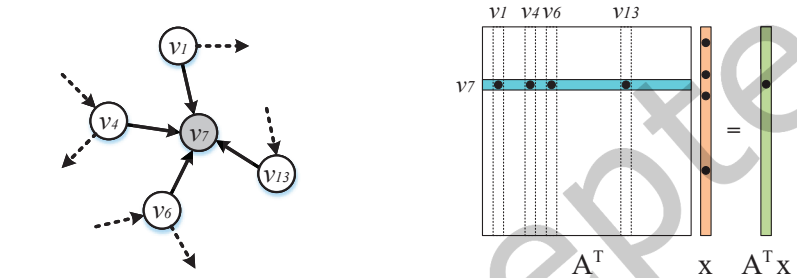
2.1 ReRAM Basics

The *resistive random access memory* (ReRAM) [49] is an emerging non-volatile memory that can rapidly perform the analog computation in-situ mode. Figure 1 shows the basics of ReRAM, where the ReRAM cell is the basic unit. Through applying an external voltage, the resistance of a ReRAM unit is able to be switched between states of *low resistance* (LR) and *high resistance* (HR), where the states of LR and HR can be used to symbolize the logic "1" and "0", respectively. The crossbar architecture is used to organize ReRAM cells efficiently and each wordline is connected to each bitline via ReRAM cells, as shown in Figure 1(b). If the input voltages applied on the wordlines are V_1, V_2, ..., V_n, the i^{th} ReRAM cell will pass the current of $V_i \times R_i$ to the bitline and the total current of the bitline will be the sum of the currents (i.e., $\sum V_i \times R_i$) passing through each ReRAM cell in the bitline, as shown in Figure 1(a). This provides the capability to perform in-situ vector-vector multiplication operation, while the *Matrix-Vector Multiplication* (MVM) operation can be supported by applying multiple bitlines with massive parallel. To support the execution of MVM, as shown in Figure 1(b), the *digital-to-analog converters* (DACs) are used to convert input digital values to analog format and transmit them to wordlines. The *sample and hold* (S&H) units are employed to extract the output column currents of bitlines and pipe these currents into the *analog-to-digital converters* (ADCs), where the ADC is used to convert the analog current into the digital format.



(a) The current on the bitline (b) ReRAM crossbar

Fig. 1. Basics of ReRAM



(a) Vertex-centric graph processing (b) SpMV-based graph processing

Fig. 2. Graph processing in different views

2.2 ReRAM-based Graph Processing

The graph G=(V, E) is able to be represented as an adjacency matrix naturally, where the rows of the matrix represents the source vertices and the columns of the matrix denotes the destination vertices. The number of rows and columns within the matrix equals that of vertices ($|V|$). Note that, because each undirected graph is able to be converted into a directed graph, we use directed graph to illustrate it. The graph algorithms typically obtain their corresponding final states of vertices through iteratively processing the vertices and edges of the graph, where the state of each vertex is usually updated by its neighbor vertices iteratively. In the popular vertex-centric graph processing model [15, 17, 43], as shown in Figure 2(a), each vertex (e.g., v_7) receives the messages from its neighbors (i.e., v_1, v_4, v_6, and v_{13}) along the corresponding in-coming edges and updates its own state to reach a state closer to the convergence value. Accordingly, in existing ReRAM-based graph processing accelerators [44, 69], as depicted in Figure 2(b), the processing of the vertices in each iteration can be implemented by a *Sparse Matrix-Vector Multiplication* (SpMV) operation (i.e., $\mathbf{A^T x}$) over the ReRAM crossbar, where the adjacency matrix $\mathbf{A}$ represents the graph, $\mathbf{A^T}$ is the transpose of $\mathbf{A}$, and the vector $\mathbf{x}$ denotes the vertices' states within the current iteration. Because of the limited size of ReRAM crossbars, the whole graph typically needs to be partitioned into a series of matrix-formatted subgraphs. Then, these subgraphs are sequentially streamed into the ReRAM crossbar for processing when an active vertex is included.

2.3 Problems of Existing Solutions

Existing ReRAM-based graph processing accelerators [5, 11, 44, 69] typically define a specific processing order (e.g., column-major and row-major) to iteratively handle the matrix-formatted subgraphs, in which each subgraph is handled at most once in an iteration. Consequently, the vertices within each subgraph typically require many iterations to propagate their new states to their successors. ReRAM-based graph processing typically takes the

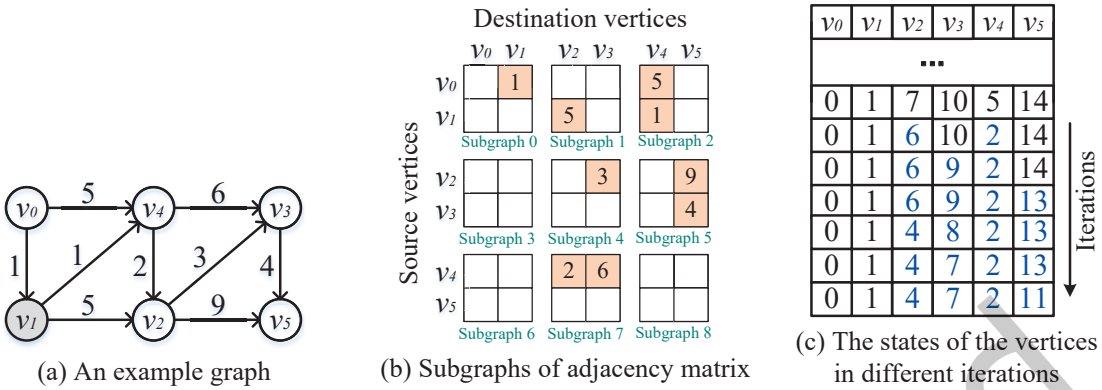


Fig. 3. An example to illustrate the inefficiency of existing ReRAM-based graph processing solutions: (a) an example graph; (b) the subgraphs of the adjacency matrix corresponding to the graph in (a); (c) the state of vertices in different iterations, where the blue value means the change of vertex state

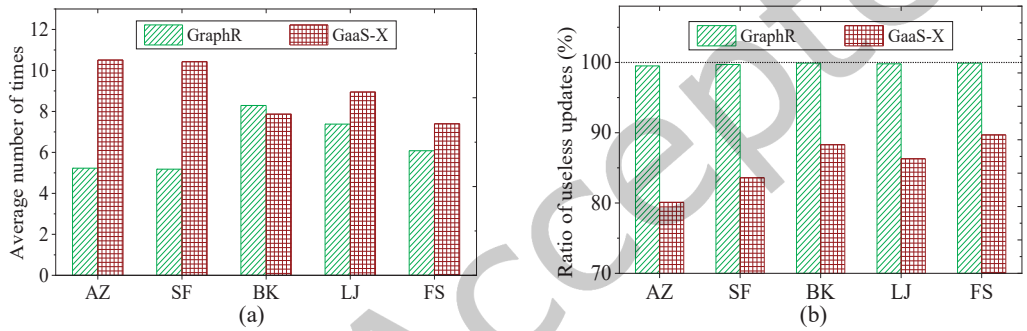


Fig. 4. The performance on existing solutions when applying the SSSP algorithm on different datasets: (a) the average number of each subgraph's processing times; (b) the ratio of useless updates against the total number of updates

matrix-formatted subgraph as the basic processing unit, which is different from CPU-based and GPU-based graph processing that take vertex/edge as the basic processing unit. Therefore, the different vertices' states in each matrix-formatted subgraph are concurrently and irregularly propagated to other matrix-formatted subgraphs. Besides, the subgraphs of the dependency chain may be concurrently handled out of order on multiple ReRAM crossbars. Thus, the vertices of these subgraphs have to iteratively calculate their states based on the stale states of the vertices corresponding to their neighbor subgraphs. As a result, existing solutions suffer from significant redundant computation overhead.

We use Figure 3 to illustrate the above problem, where the example graph is partitioned into nine matrix-formatted subgraphs. As shown in Figure 3(b), each subgraph contains two source vertices and two destination vertices, and the orange region represents the valid element in the matrix. We assume that there is one active vertex v_1 (the gray vertices in Figure 3(a)) in the current iteration. With existing ReRAM-based approaches [11, 44, 69], the new state of v_1 can only be propagated to its direct neighbors (i.e., the vertices v_2 and v_4) in each iteration, because each new vertex state cannot be used by other vertices in the same iteration. In order to propagate the new state of v_1 to v_3 and v_5, two iterations of graph processing are required. Besides, a graph path (e.g., $v_1 \rightarrow v_4 \rightarrow v_2 \rightarrow v_3 \rightarrow v_5$) may be partitioned into different subgraphs, where there are dependency chains exist among these subgraphs. However, the subgraphs of the dependency chains may be concurrently handled by different ReRAM crossbars within each iteration. The whole graph has to be handled for many iterations to propagate each vertex's state to the others along dependency chain, because the state of an already-processed neighbor can only

be updated in the next iteration. For example, as shown in Figure 3(c), after the processing of Subgraph 1 and Subgraph 2, the states of v_2 and v_4 are updated to 6 and 2, respectively. After that, Subgraph 4, Subgraph 5, and Subgraph 7 will be handled by the ReRAM crossbars. In detail, for the processing of Subgraph 4 and Subgraph 5, v_2 propagates its state to v_3 and v_5, and then updates their states to 9 and 13, respectively. For the processing of Subgraph 7, the states of v_2 and v_3 will be updated using v_4's state. Therefore, Subgraph 4 and Subgraph 5 have to be loaded and handled again according to the newest states of v_2 and v_3. That is to say, the computation of Subgraph 4 and Subgraph 5 according to the state of v_2 (i.e., 6) are useless when v_4's new state (i.e., 2) has not to be propagated to v_2. Note that although GaaS-X can handle v_2 and v_4 concurrently through gathering and applying operations for a single vertex, it still suffers from redundant computations. For example, when v_3's state has been updated to 8 according to both v_2's new state (i.e., 6) and v_4's new state (i.e., 2), v_2's state will be simultaneously updated to 4 based on v_4's state (i.e., 2). Thus, v_3 has to be loaded and handled again according to v_2's newest state (i.e., 4). This indicates that it is useless to calculate v_3's state according to v_2's state (i.e., 6) and v_4's state (i.e., 2), because v_2's state will be updated according to v_4's new state and the dependency chain between v_4 and v_2.

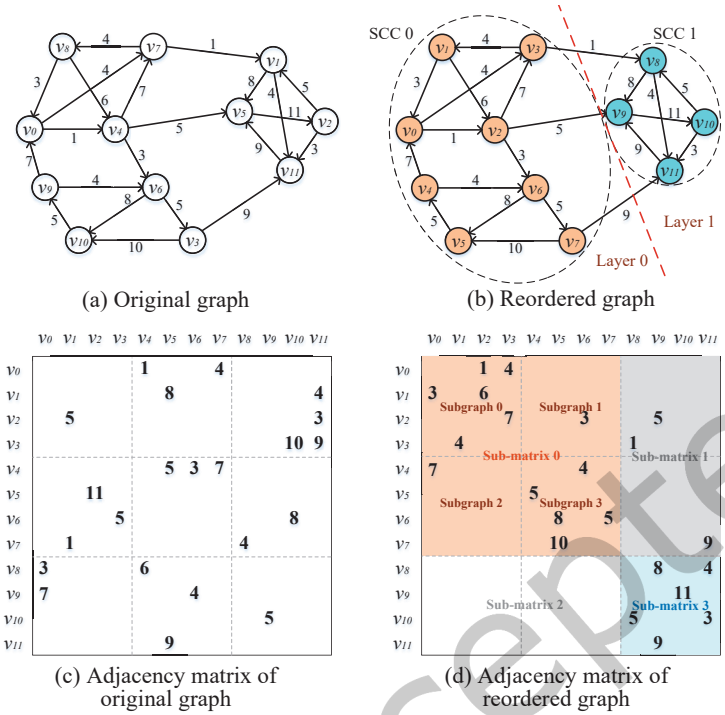
To demonstrate it, we evaluate two cutting-edge ReRAM-based graph processing accelerators (i.e., GraphR [44] and GaaS-X [5]) by running the *Single Source Shortest Path* (SSSP) algorithm. The platform and benchmarks are detailed in Section 4. As shown in Figure 4, existing solutions suffer from significant redundant computation overhead, which results in the underutilization of the ReRAM crossbar arrays. Although GaaS-X needs less computations through alleviating the computations on the zero valued edges, it still performs many redundant computations. It is because the subgraphs of the dependency chains may also be concurrently handled by several ReRAM crossbar arrays. Consequently, the vertices' states of many subgraphs are updated according to the stale vertex states in their neighbors for several iterations before receiving the most recent vertex states from their neighbors, and need to be reprocessed. As shown in Figure 4(a), we can observe that many subgraphs need frequent reprocessing, because each vertex usually reads stale states of its neighbors for useless update before updating these stale states in the same iterations. GaaS-X requires more reprocessing than GraphR, because GaaS-X supports the native sparse representation of graph data and thus each subgraph of GaaS-X contains more edges than that of GraphR. It indicates that more state propagations will pass through each subgraph in GaaS-X. Figure 4(b) shows that the number of useless vertex state updates occupies more than 80.1% of that of total vertex state updates in GaaS-X. GaaS-X conducts fewer useless updates than GraphR because GaaS-X can alleviate the unnecessary computations on zero-valued edges.

2.4 Motivation

We have the following two observations regarding the redundant computation overhead in ReRAM-based graph processing.

Observation one: When constructing the subgraph with good internal connectivity through tracking the dependencies between the active vertices and their successors, the new vertex states within this subgraph can be able to quickly and efficiently work on their successors using an asynchronous way. Taken Figure 5(a) as an example, we assume a subgraph contains four vertices and the active vertex is v_0. When we track the dependencies between v_0 and its successors, we can construct a tightly connected subgraph, which is composed of v_0, v_4, v_7, and v_8. In this way, only the active vertices and their successors need to be loaded for processing, avoiding the unnecessary computation overhead. Besides, when we asynchronously handle each tightly connected subgraph iteratively, it enables the new vertex state (e.g., the state of v_0 in Figure 5(a)) with this subgraph to be quickly propagate to its successors (i.e., v_4, v_7, and v_8 in Figure 5(a)) in the same subgraph.

Observation two: Most redundant computations of processing the subgraphs over the ReRAM crossbars can be alleviated when handling these subgraphs along the dependencies among their vertices. As shown in Figure 3(a), after



(a) Original graph (b) Reordered graph

(c) Adjacency matrix of original graph (d) Adjacency matrix of reordered graph

Fig. 5. An example to illustrate dependency-aware subgraph construction: (a) original graph; (b) the reordered graph; (c) the adjacency matrix of the original graph; (d) the adjacency matrix of the reordered graph

the processing of active vertex v_1, the states of v_2 and v_4 are updated to 6 and 2, respectively. Then, the subgraphs containing v_2 and v_4 need to be processed. The processing order of these subgraphs affects the convergence rate of the graph algorithm. Specifically, if we preferentially process subgraph containing v_4, v_2 receives the state propagation from v_4 and updates its own state to 4. After that, the subgraph containing v_2 is assigned to be handled. Then, it can directly update the state of v_3 and v_5 using the latest state of v_2 (i.e., 4). As a result, the useless state updates of v_3 and v_5 according to the state of v_2 (i.e., 6) are avoided. In this way, the vertices (e.g., v_3 and v_5) of the subgraphs need much fewer vertex state updates, reducing the redundant computation overhead significantly.

3 Overview of Our Solution

Based on the above observations, we propose an efficient ReRAM-based accelerator *ASGraph*, using our proposed *dependency-aware asynchronous processing model* to achieve efficient execution of graph algorithms. In this section, we first present the main idea of our proposed processing model and then detail the implementations of ASGraph.

3.1 Dependency-aware SpMV-based Graph Processing Model

This subsection will introduce the main idea of the *dependency-aware asynchronous processing model*, which dynamically constructs the matrix-formatted subgraphs based on dependencies between vertices, regularizes the state propagations among these constructed subgraphs, and applies a hybrid processing scheme to minimize redundant computation overhead. The details of our method are described below.

Algorithm 1 Graph Translation Algorithm

Input: Original graph $G = (V, E)$

1: $DAG \leftarrow$ `ConstructDAG`(G) ▷ Construct the DAG of the graph
2: **while** `IsNotEmpty`(DAG) **do** ▷ DAG is not empty
3: $Layer \leftarrow$ `GetAllSCCHasNoInDegree`(DAG) ▷ Get all SCCs without in-degree
4: `DeleteAllSCCinLayer`$(DAG, Layer)$ ▷ Delete all SCCs in current Layer
5: `Ascending`$(Layer)$ ▷ Ascending layer
6: **end while**
7: $G' \leftarrow$ `Reorder`$(G, DAG, Layer)$ ▷ Reorder the vertices of G to get the reordered graph G'

Dependency-aware Subgraph Construction. Based on our observation, we can alleviate the irregular state propagations through handling the graph data according to the graph topology. Thus, as shown in Algorithm 1, we first represent the original graph as a directed acyclic graph (DAG) [62] through extracting *Strongly Connected Components* (SCCs)[3] using Tarjan algorithm [45] (line 1). After that, as shown in Figure 5(b), the SCCs of DAG are divided into layers according to their topological order in the DAG (lines 2-6), where the SCC in a layer can only be activated by the SCC at the lower layer. Then, we reorder the vertices following in the topological order of the SCCs in the DAG (line 7), where the vertices inside the same SCC are consecutively arranged, as shown in Figure 5(b). Figure 5(d) represents that the adjacency matrix of this reordered graph, where this adjacency matrix consists of three parts. The first part is the sub-matrices on the diagonal (e.g., the sub-matrix 0 and sub-matrix 3 in Figure 5(d)), which are the SCCs of the graph. The second part is the elements above the sub-matrices on the diagonal (e.g., the sub-matrix 1 in Figure 5(d)), which correspond to the edges among SCCs. The third part is the elements below the sub-matrices on the diagonal (e.g., the sub-matrix 2 in Figure 5(d)). Due to the SCCs do not constitute a cycle, there are no valid elements in the third part. During the processing, we can iteratively handle the sub-matrices on the diagonal along the topological order between their corresponding SCCs for regular state propagations. That is, sub-matrix 3 is assigned to be handled only when the vertices (i.e., v_0, v_1, ... , and v_7 in Figure 5(d)) associated with sub-matrix 0 have converged. Note that the elements above the sub-matrices on the diagonal (e.g., sub-matrix 1) only need to be handled once for the propagation of the new vertex states among SCCs (i.e., sub-matrix 0 and sub-matrix 3).

For the processing of each SCC, it usually needs to be divided into a series of matrix-formatted subgraphs with specified size (which is determined by the size of the ReRAM crossbar). The vertex state is inherently propagated along the dependencies among the vertices, thus only active vertices and their successors need to be processed during the execution. Based on this observation, we dynamically extract a set of the tightly connected vertices through tracking the dependencies between these active vertices and their successors, and then uses these vertices' edges in this SCC to construct several matrix-formatted subgraphs that need to be handled. Note that the number of vertices extracted each time is determined by the crossbar size. In this way, the new vertex states can be propagated quickly during the processing of each subgraph.

Specifically, as shown in Algorithm 2, it first adds an unvisited active vertex into the vertex set S of this subgraph (lines 1-3), and then explore the optimal vertex from the neighbors of the vertices in S (lines 5-7). This procedure will be repeated until the size of the subgraph reaches the crossbar size or all neighbors of the vertices in S have been added (line 4). `OptimalNeighbor` is employed to obtain the neighbor that is most densely connected to the vertices in S. Note that the optimal neighbor v_j satisfies the condition $\texttt{Connect}(v_j) = \text{argmax}_{v \in Neighbor(S)} \texttt{Connect}(v)$, where $\texttt{Connect}(v)$ is used to count the number of edges between the vertices in

[3]Each SCC represents a maximal subgraph in which every pair of vertices is reachable from each other.

Algorithm 2 Dependency-aware Subgraph Construction

Input: Reordered graph $G' = (V', E')$, active vertex set V_a, the set of the tightly connected vertices S, ReRAM crossbar size C_size

```
for each unvisited vertex v_i ∈ V_a do                ▷ Explore each unvisited active vertex
    Set v_i as visited
    S ← {v_i}
    while Size(S) < C_size and there is unvisited neighbor of S do
        v_j ← OptimalNeighbor(S)                      ▷ Obtain the optimal vertex v_j from S's neighbors
        Set v_j as visited
        S ← S ∪ {v_j}                                 ▷ Add v_j into S
    end while
    ConstructSubgraph(S)                              ▷ Construct the matrix-formatted subgraph using S
end for
```

S and the vertex v. After that, the edges of the vertices in S are used to construct the matrix-formatted subgraphs, where the edges between the tightly connected vertices will be constructed into the same subgraph.

Taken Figure 5 as an example, where the active vertex is assumed as v_0 in the reordered graph and the crossbar size is assumed as four. v_0, v_1, and v_2 will be sequentially added into S. Next, the neighbors of S are v_3, v_6, and v_9. Because v_3 has the largest number of edges that are connected to the vertices in S, v_3 will be added into S, so that the set of the tightly connected vertices contains v_0, v_1, v_2, and v_3. Then, these vertices' edges (i.e., the non-zero elements in sub-matrix 0) is used to construct the subgraph 0 and subgraph 1, where subgraph 0 consists of the edges between the tightly connected vertices. By such means, the new vertex state (e.g., v_0) can be immediately and efficiently used to update the states of its successors (e.g., v_1, v_2, and v_3) on the same subgraph (i.e., subgraph 0) in the same iteration, indicating faster convergence rate than existing solutions.

Value-driven Subgraph Scheduling. For the processing of each SCC, a wrong execution order of its subgraphs may result in massive redundant computations. It is because the vertices' states of many subgraphs may be frequently updated according to the stale vertex states of their neighbors within the same SCC. To overcome this limitation, we propose a value-driven scheduling method to adaptively assign the processing order of subgraphs, ensuring a fast convergence speed. Specifically, unlike existing solutions [59, 68] mainly use graph structural characteristics to determine priority, our approach specifies a subgraph with a higher value when the vertices of this subgraph have accumulated a lot of state propagations from their neighbors (i.e., the greater the change in its vertex states) or can affect the vertex states of more subgraphs (i.e., its vertices has more neighbors). These rules can be expressed as $Value_j = \sum_{v_i \in V_a} |s_i^{k_i} - s_i^{k_i-1}| \times \lg(d_i + 1)$, where $Value_j$ is the specified value of the j^{th} subgraph, V_a is the set of active vertices, $s_i^{k_i}$ and $s_i^{k_i-1}$ are the states of v_i in the k_i^{th} and $(k_i - 1)^{th}$ iterations, respectively, and d_i is the outdegree of v_i. Then, the subgraphs with the higher values will be preferentially mapped to the ReRAM crossbars for processing. Note that the subgraphs with the same set of source vertices will be given the same value, thus we apply the row-major processing order to handle the subgraphs within each SCC and the processing order of the rows of subgraphs is determined by their values. Taken Figure 3 as an example, where v_2 and v_4 are activated when v_1 has propagated it's new state to them. Then, the subgraphs containing v_2 and v_4 (i.e., Subgraph 4, Subgraph 5, and Subgraph 7) need to be processed. Because the change of v_4's state is larger, the subgraphs containing v_4 (i.e., Subgraph 7) will be preferentially handled. After that, v_4 propagates its new state to v_2 and v_3 and the state of v_2 can be instantly updated from 7 to 4. Thus, the redundant state propagations associated with the stale state of v_2 can be eliminated.

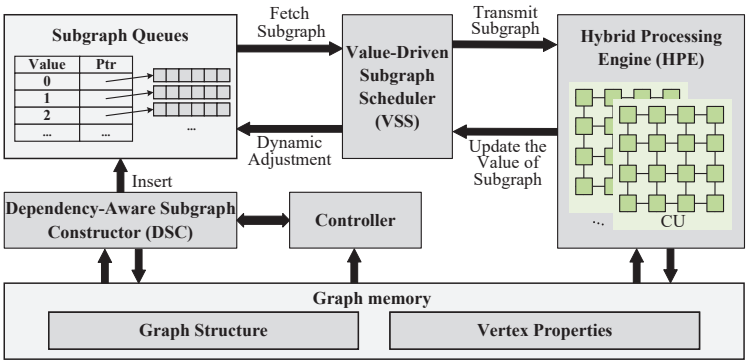


Fig. 6. Overview of ASGraph design

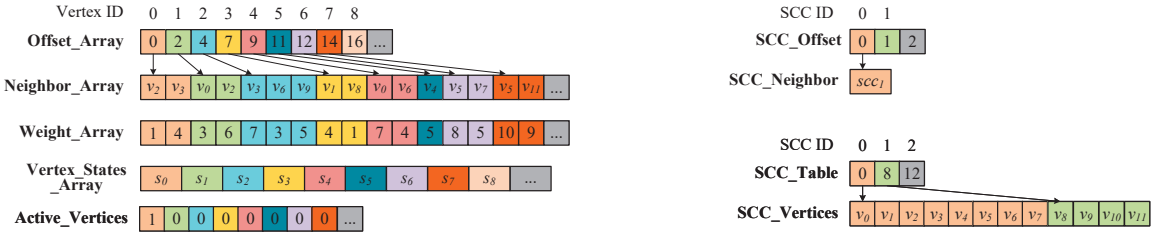
Hybrid Processing Scheme. For the processing of each row of subgraphs (i.e., the ones with the same set of source vertices and the same value), we design a hybrid processing scheme to handle these subgraph to minimize the redundant computations. Specifically, the subgraph consisting of edges between tightly connected vertices (e.g., subgraph 0 in Figure 5(d)) is preferentially assigned to be iteratively processed until its vertices' states are unchanged, enabling the new state of the vertices to be propagated quickly. After that, the other subgraphs (i.e., subgraph 1 in Figure 5(d)) of the same row will be assigned to be handled once. In this way, it makes the states of the tightly connected vertices closer to the convergence states before these vertices propagate their states to other vertices (i.e., the source vertices of other rows), minimizing the redundant computations.

3.2 Overview of ASGraph

Based on our *dependency-aware SpMV-based graph processing model*, we propose a novel ReRAM-based asynchronous graph processing accelerator *ASGraph*. Figure 6 shows the overview of ASGraph, which consists of three primary components, i.e., *Dependency-Aware Subgraph Constructor* (DSC), *Value-Driven Subgraph Scheduler* (VSS), and *Hybrid Processing Engine* (HPE). Different from existing ReRAM-based accelerators [5, 11, 44, 69], the DSC is used to dynamically construct the matrix-formatted subgraphs on the fly according to the dependencies among the vertices' states. After that, the VSS of ASGraph is employed to assess the value of the constructed subgraphs and preferentially assign high-value subgraphs to be handled by the HPE. The HPE of ASGraph consists of a number of customized ReRAM-based crossbar arrays, which employ a hybrid processing scheme to handle each row of subgraphs within the same SCC. Note that ASGraph relies on the host to represent the original graph as a DAG, reorder the vertices along their topological order shown in the DAG, and divide the reordered graph (e.g., the graph in Figure 5(b)) into a series of sub-matrices as shown in Figure 5(d). The Controller of ASGraph assigns these sub-matrices to be handled according to their topological order (i.e., the order should be sub-matrix 0, sub-matrix 1, and sub-matrix 3), where each sub-matrices only needs to be assigned once. The main functionalities of the components of ASGraph are as follows.

When Controller assign a sub-matrix according to the topological information shown in the DAG, the DSC starts from each active vertex to track the dependencies between this vertex and its successors so as to extract the set of vertices that are tightly connected with it. After that, it use the edges of these extracted vertices to construct the subgraphs and then insert these constructed subgraphs into the *Subgraph Queues*. In this way, it enables the fast state propagation for each active vertex when processing its corresponding subgraph over the ReRAM crossbar array.

When the subgraphs have been constructed in the *Subgraph Queues*, the VSS of ASGraph will calculate their values for achieving value-driven scheduling. Besides, the vertices' states in the subgraphs will be updated at



(a) Graph storage in the CSR format (b) The representation of the corresponding DAG

Fig. 7. The data structure of ASGraph (corresponding to the graph in Figure 5)

runtime during the processing, VSS needs to dynamically maintain the values of the constructed subgraphs. During the processing, VSS will obtains the subgraphs with the highest value from the *Subgraph Queues*, where the obtained subgraphs are stored in the VSS's buffer and will be transmitted to the HPE for processing. Note that each row of subgraphs in the assigned sub-matrix is given the same value, thus this value only needs to be calculated once for multiple subgraphs to reduce the calculation overhead. Meanwhile, it means that each row of subgraphs will be transmitted to the HPE together.

When the HPE receives the subgraphs (i.e., the subgraphs within the same row) from VSS, HPE preferentially takes out the tightly connected subgraph (i.e., its source vertex set is the same as its destination vertex set) and convert its format to the adjacency matrix format for processing. After that, HPE will load this subgraph into the *Computing Unit* (CU) and iteratively handle it until its vertices' states don't change anymore. After that, the other subgraphs of the same row will be assigned to be handled by HPE, where these subgraphs only need to be processed by CU once. By such means, the corresponding source vertices of each row can accumulate more state propagations originated from each other and then propagate these new states to other vertices (i.e., the vertices of other rows), further reducing the redundant computations. Note that, after the processing of each subgraph, HPE will update the corresponding vertex states and triggers VSS to update the values of the constructed subgraphs according to their latest vertex states.

3.3 Hardware Design

3.3.1 Key Data Structure of ASGraph: In this subsection, we present the key data structures required by the ASGraph. Because *Compressed Sparse Row* (CSR) is the most popular format, like existing ReRAM-based solutions [69], ASGraph uses the CSR format to store the graph data as shown in Figure 7(a). Specifically, it mainly uses three arrays to store the graph, i.e., *Offset_Array*, *Neighbor_Array*, and *Weight_Array*. The begin/end offset of neighbor vertices are stored in *Offset_Array*. *Neighbor_Array* stores the neighbor vertex's ID corresponding to each outgoing edge. *Weight_Array* records the weight of each edge. We also record the states of vertices in *Vertex_States_Array*. Meanwhile, as shown in Figure 7(a), a bitvector *Active_Vertices* is used to record the active vertices. Note that we store the reordered graph using the above format for ASGraph. Besides, we also store the DAG of the reordered graph as shown in Figure 7(b). To efficiently store the DAG, *SCC_Offset* stores the offsets of the beginning and end of the neighbors for each SCC, and *SCC_Neighbor* maintains all neighbors for them. *SCC_Table* and *SCC_vertices* are employed to record the set of vertices contained in each SCC. Note that an array, i.e., *Connectivity_Array*, is used to maintain the intermediate information during dynamically constructing the subgraphs.

3.3.2 Dependency-aware Subgraph Constructor: To construct the subgraphs, a fix-depth hardware stack is used in DSC to maintain the information of explored vertices, where the ID of each explored vertex and the current and end offsets of its neighbors are stored in a level of the stack. The depth of the stack is determined by the ReRAM crossbar size. Note that *Connectivity_Array* is used store all neighbors of the vertices in the stack and also record

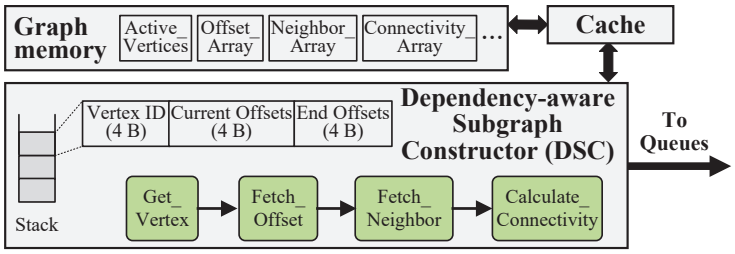


Fig. 8. The pipeline of DSC

the number of the edges from these neighbors to the vertices in the stack. As shown in Figure 8, to efficiently realize dependency-aware subgraph construction, four stages, i.e., *Get_Vertex*, *Fetch_Offset*, *Fetch_Neighbor*, and *Calculate_Connectivity* are implemented as a pipeline. Specifically, at the beginning of the pipeline, if the stack is empty, the *Get_Vertex* stage gets an unvisited active vertex by scanning *Active_Vertices* and pushes it into the stack. Otherwise, the *Get_Vertex* stage selects the neighbors with the maximum number of connected edges from the *Connectivity_Array* and pushes it into the stack. In the *Fetch_Offset* stage, DSC fetches the beginning and end offset of its neighbors from the *Offset_Array*. In the *Fetch_Neighbor* stage, the current offset of the top vertex in the stack is used to fetch the unvisited neighbor vertices of it, and then these fetched neighbors are set as visited. In the *Calculate_Connectivity* stage, the value of the fetched neighbor vertex in the *Connectivity_Array* is increased by one, which indicates that the number of edges from the vertices in the stack to this fetched neighbor vertex is increased by one. At the end of this pipeline, if the number of the vertices in the stack reaches the ReRAM crossbar size, all the vertices in the stack are popped and used to construct a subgraph, which is stored in the *Subgraph Queues*. Then, the values in the *Connectivity_Array* will be initialized to zero and repeats the above process until all active vertices have been visited.

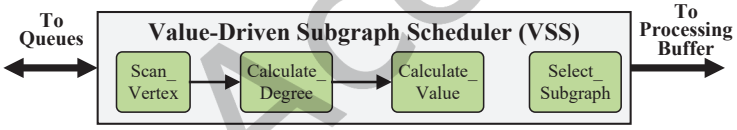


Fig. 9. The pipeline of VSS

3.3.3 Value-Driven Subgraph Scheduler. When some subgraphs are constructed by the DSC, the VSS of ASGraph needs to calculate the values of the subgraphs for scheduling. Specifically, as shown in Figure 9, three stages, i.e., *Scan_Vertex*, *Calculate_Degree*, and *Calculate_Value* are implemented as a pipeline. The *Scan_Vertex* stage scans the source vertices of each subgraph in the *Subgraph Queues* to obtain their states and neighbors' beginning and end offsets. After that, at the *Calculate_Degree* stage, it subtracts each scanned vertex's beginning offset from its end offset to obtain the outdegree of this vertex. Based on these information (i.e., the states and outdegrees of the vertices in this subgraph), the *Calculate_Value* stage calculates the value for this subgraph according to the rules defined in Section 3.1. Note that the values of the subgraphs that have the same set of source vertices only need to be calculated once. Then, the *Select_Subgraph* logic of VSS requests the subgraphs with the highest value from the the *Subgraph Queues* and then transmits these subgraphs to the *Processing Buffer* of HPE to drive the processing of the HPE. Note that the states of vertices in the subgraphs may be updated during the execution, thus VSS will dynamically update the values of the corresponding subgraphs according the state updates of the vertices.

3.3.4 Hybrid Processing Engine. When the *Processing Buffer* of HPE receives the subgraphs from VSS, HPE will preferentially handle the tightly connected subgraph and then handle the other subgraphs with the same set of source vertices, minimizing the redundant computations. Specifically, the prefetcher preferentially assigns the

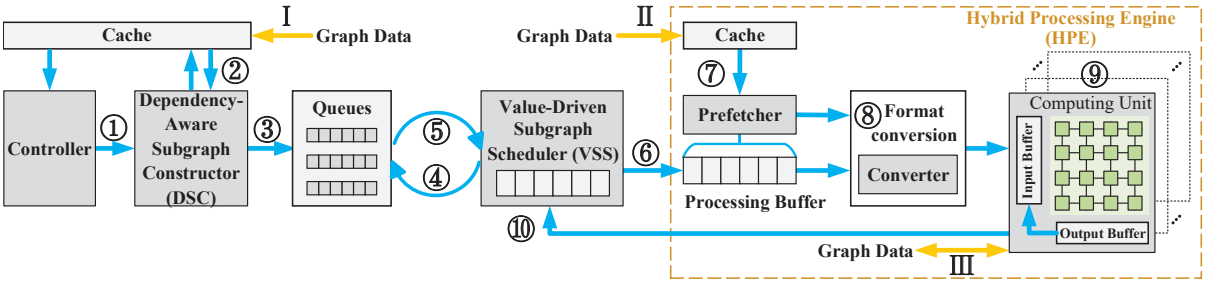


Fig. 10. Execution flow of ASGraph, where the blue arrows represent the data flow and the yellow arrows represent memory access

tightly connected subgraph to the converter. After that, the converter converts the graph data of the assigned subgraph in CSR format to the format of adjacency matrix and then maps this adjacency matrix on the ReRAM crossbar of a CU. Meanwhile, the prefetcher loads the vertex states associated with this assigned subgraph and sends these states into the input buffer for processing. Note that each CU is assigned to process the subgraphs with the same source vertices, while different CUs handle different row of subgraphs for parallelism. For the processing of the tightly connected subgraph over each CU, this subgraph needs to be iteratively handled by the this CU until its vertices' states are unchanged, i.e., the previous vertex states in the input buffer is the same as the latest vertex states in the output buffer. If these vertices' states are changed, the CU can directly use the vertex states in the output buffer as the input for iteratively processing. Otherwise, the other subgraphs with the same set of source vertices of this tightly connected subgraph will be assigned to be processed by this CU once. The corresponding vertex state updates will be send to the VSS to update the values of the corresponding subgraphs. Besides, the new active vertices will also be labeled in the *Active_Vertices*, which is used to trigger DSC to construct the new subgraphs.

3.3.5 Execution Flow: Figure 10 shows the execution flow of ASGraph. Specifically, the Controller of the ASGraph first assigns a sub-matrix to be handled according to its topological order shown in the DAG of the graph (step ①). After that, DSC starts from each active vertex to traverse the graph structure data (i.e., step I) so as to construct the tightly connected subgraph in the assigned sub-matrix through tracking the dependencies among this active vertex and its successors (step ②), and also maintains them in the *Subgraph_Queues* (step ③). For the constructed subgraphs, VSS calculate their values (step ④) and uses these values to achieve subgraph scheduling (step ⑤), where the subgraphs with the largest value will be preferentially assigned for processing (step ⑥). When a subgraph is assigned to the *Processing Buffer*, HPE prefetches the vertex states of these subgraphs (step ⑦ and step II) and also converts the format of these subgraphs to the adjacency matrix for mapping (step ⑧). When a subgraph is mapped over the CU of the HPE, it will be handled by this CU accordingly (step ⑨). After the processing of each subgraph, the corresponding vertices' states may be updated, where these vertex state updates need to be send to the VSS (step ⑩) and these updated vertex state data will also be applied in *Vertex_States_Array* (i.e., step III). VSS will use these vertex state updates to update the values of the corresponding subgraphs (step ④).

4 Evaluation

4.1 Experimental Setup

The cycle accurate simulator based on Structural Simulation Toolkit [38] is used in our experiments to model the main hardware units, including Controller, *Dependency-Aware Subgraph Constructor* (DSC), *Value-Driven Subgraph Scheduler* (VSS), and the control unit in *Hybrid Processing Engine* (HPE). For the *Computing Unit* (CU) in HPE, we

Table 1. Hardware Specifications of ASGraph

Component	Area ($mm^2 \times 10^{-3}$)	Power (mW)	Parameters	Specifications
CU Properties (2048 CUs)				
ReRAM crossbar	1.58	11.80	Size	8×8
			Number	1 Per CU
DAC	0.4	2.80	Number	8 Per CU
S&H	0.02	0.024	Number	8 Per CU
ADC	33.80	31.20	Number	1 Per CU
SFU	11.50	1.21	Number	1 Per CU
Other Properties				
Subgraph Constructor	212.00	10.10	Latency	112 ns
Subgraph Scheduler	137.60	8.24	Latency	33 ns
Prefetcher	21.20	2.05	Latency	24 ns
Converter	15.52	1.74	Latency	20 ns
Controller	116.80	6.52	Latency	16 ns
Subgraph Queues	102.40	139.52	Size	256 KB
Input/Output Buffer	51.20	69.76	Size	128 KB
Processing Buffer	3.20	4.36	Size	8 KB
Cache	409.60	558.08	Size	1024 KB

Table 2. Configuration of the CPU

CPU	Intel(R) Xeon(R) Platinum 8358 CPU, 64 cores, x86-64 ISA, 2.6 GHz
L1 Instruction Cache	32 KB per-core, 8-way set-associative
L1 Data Cache	48 KB per-core, 12-way set-associative
L2 cache	1280 KB, 20-way set-associative
L3 cache	48 MB, 12-way set-associative
Memory	1 TB

adopt the architecture of ReRAM crossbar to implement it. Besides, NVSim [13] is used to estimate the execution time and energy consumption of the ReRAM crossbar in our experiments. The specific parameter information of the ReRAM cell reported in [33] is as follows. The LRS/HRS resistance of the ReRAM cell are 50 KΩ and 25 MΩ and the write/read voltage are 2 V and 0.7 V. In addition, the write/read latency are 50.88 ns and 29.31 ns, and the write/read energy consumption are 3.91 nJ and 1.08 pJ, respectively. The data of Analog/Digital converters are from [30], and the precision of ADCs and DACs are 6 bits and 2 bits. Table 1 summarizes the ASGraph configurations and its area-energy breakdowns, where each *Special Function Unit* (SFU) consists of *shift and add units* (S&A) and *scalar arithmetic and logic unit* (sALU) to process the crossbar outputs and perform additional operations to support various graph algorithms. Note that the latency and power consumption parameters of the on-chip buffers (e.g., *Subgraph Queues*, *Processing Buffer, and Input/Output Buffer*) are modeled using CACTI 6.5 [1] at 32 nm scale. The control circuitry (e.g., *DSC*, *VSS*, *Prefetcher*, and *Converter*) is implemented using SystemVerilog RTL, and the power consumption and area metrics are obtained by doing RTL synthesis in 32 nm technology for operation at 1 GHz. There are 128 queues in the *Subgraph_Queues* and 1024 entries in the *Processing Buffer* in our experiments by default.

Table 3. Configuration of the GPU

Graphic Card	NVIDIA A100-SXM4-80GB
CUDA cores	6912
Base Clock	1065 MHz
Graphic Memory	80GB HBM2e
Memory Bandwidth	2039 GB/s
CUDA Version	11.7

Table 4. Characteristic Statistics of Datasets

Datasets	Vertices	Edges	Avg Degree
Amazon(AZ) [21]	262,111	1,234,877	9
web-Stanford (SF) [21]	281,904	2,312,497	16
web-BerkStan (BK) [21]	685,231	7,600,595	22
soc-LiveJournal (LJ) [21]	4,847,571	68,993,773	28
com-Friendster (FS) [21]	65,608,366	1,806,067,135	55

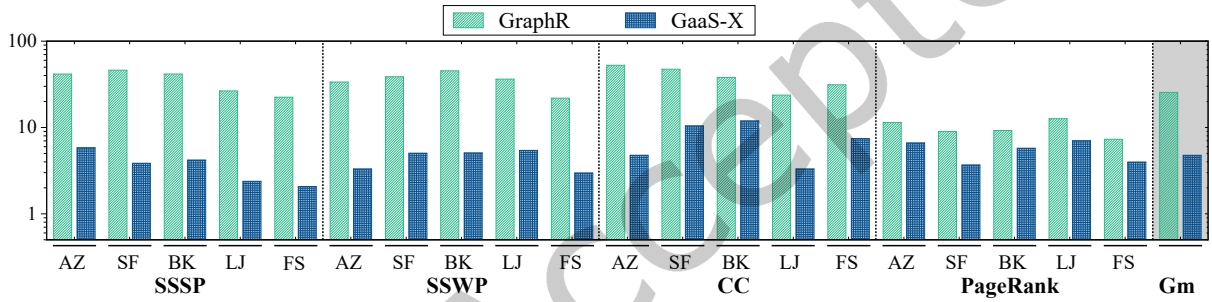


Fig. 11. Speedup compared to the cutting-edge ReRAM-based accelerators

To evaluate the effectiveness of our approach, we compare the performance of ASGraph with the best-performing shared-memory software graph processing system Ligra (v1.5) [43] and asynchronous graph processing system HotGraph (v1.0) [59] over a CPU platform with a 64-core Intel Xeon CPU (detailed in Table 2), the cutting-edge GPU graph processing systems Gunrock (v2.0.0) [48] and Scaph (v1.0) [68] over a GPU platform with 6912 CUDA cores (detailed in Table 3), and two state-of-the-art ReRAM-based graph processing accelerators (i.e., GraphR [44] and GaaS-X [5]). We use the simulators mentioned above to model GraphR and GaaS-X and keep the same crossbar size and the same number of crossbars in GraphR, GaaS-X, and ASGraph, respectively. Note that the ReRAM crossbar size and the number of ReRAM crossbars are 8×8 and 2048 in our experiments, respectively. Similar to existing works [11, 69], ASGraph, GraphR, and GaaS-X maintain graph metadata in ReRAM. The CPU and GPU power consumption are measured by using CPU Energy Meter [29] and Nnvidia-smi [34], respectively.

Evaluation Benchmarks. We evaluate the performance of ASGraph by conducting experiments on five real-world graph datasets. The characteristics of these graph datasets are shown in Table 4. In experiments, we evaluate the ASGraph using four representative graph algorithms (i.e., *Single Source Shortest Path* (SSSP) [44], *Single Source Widest Path* (SSWP) [12], *Connected Components* (CC) [69], and *Incremental PageRank* (PageRank) [58]).

4.2 Comparison to ReRAM-based Accelerators

The performance comparison of ASGraph with the cutting-edge ReRAM-based accelerators is shown in Figure 11. The experimental result shows that ASGraph outperforms GraphR [44] and GaaS-X [5] by 7.3×~52.6× and 2.1×~11.9×, respectively. Figure 12 presents the energy saving of ASGraph compared to other ReRAM-based

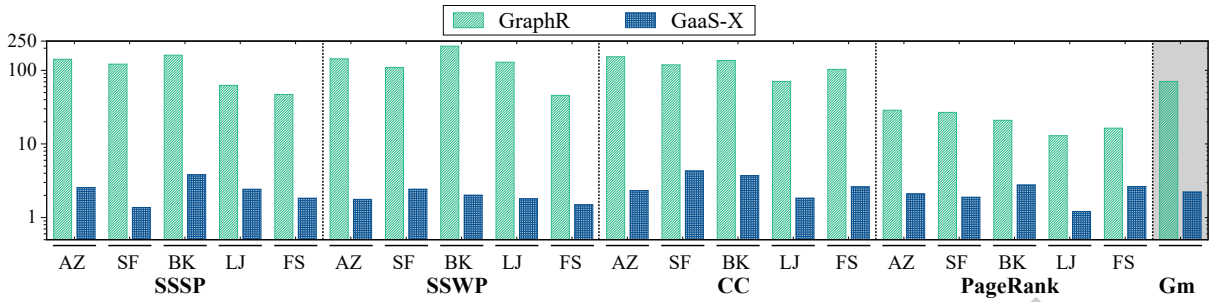


Fig. 12. Energy saving compared to the cutting-edge ReRAM-based accelerators

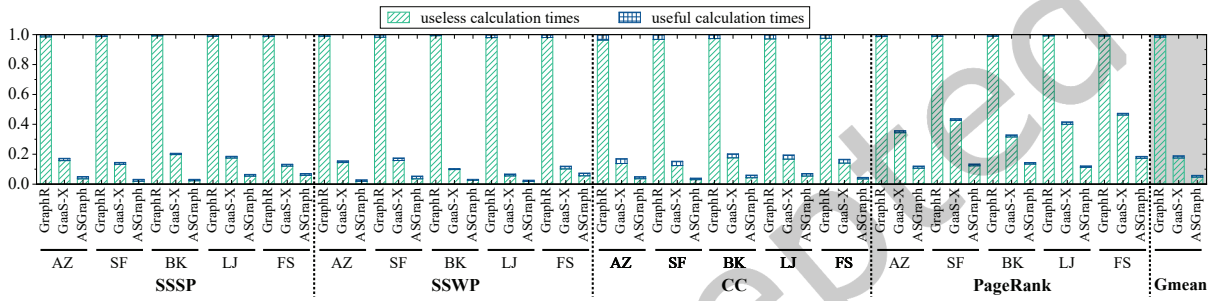


Fig. 13. Matrix calculation times normalized to that of GraphR

accelerators. It shows that ASGraph consumes 12.9×~212.8× less energy than GraphR and 1.2×~4.3× less energy than GaaS-X. Overall, ASGraph achieves an average of 25.5× speedup and 70.8× energy saving over GraphR and achieves an average of 4.8× speedup and 2.2× energy saving over GaaS-X. Note that when handling undirected graphs (i.e., the edges in the real-world graphs used in our experiments are treated as undirected edges), ASGraph still outperforms GraphR and GaaS-X by 21.4× and 4.1× on average, respectively. These performance improvements of ASGraph mainly come from the following reason.

We evaluate the breakdown of the matrix calculation times of GraphR, GaaS-X, and ASGraph, respectively. Figure 13 shows that ASGraph reduces the matrix calculation in GraphR by 81.4%~97.3% and in GaaS-X by 52.2%~79.2%, respectively. Compared to GraphR, although GaaS-X achieves a 77.2% reduction in matrix calculation by processing graphs in a sparse way, it suffers from the problem of significant redundant computation overhead. In comparison, ASGraph dynamically constructs the subgraph based on dependencies between vertices' states, regularizes the state propagations among the subgraphs, and applies a hybrid processing scheme to minimize the redundant computations, reducing the matrix operations in GraphR by 92.7% and in GaaS-X by 67% on average, respectively. Besides, as shown in Figure 13, ASGraph requires much less useless matrix calculations (i.e., only 5.2% in GraphR and 23.1% in GaaS-X on average, respectively). It means that ASGraph needs to load much fewer graph data to construct the tightly connected subgraph for processing on the ReRAM crossbars. The results show that the volume of memory access in ASGraph is only 20.5% and 21.4% of that of GraphR and GaaS-X on average, respectively.

4.3 Benefit Breakdown

We mainly proposed three optimization strategies, including *Dependency-Aware Subgraph Construction* (SC), *Value-Driven Subgraph Scheduling* (SS) and *Hybrid Processing Scheme* (HP). Figure 14 evaluates the effectiveness of these three stages respectively, where ASGraph-without is the basic version of ASGraph that uses only the

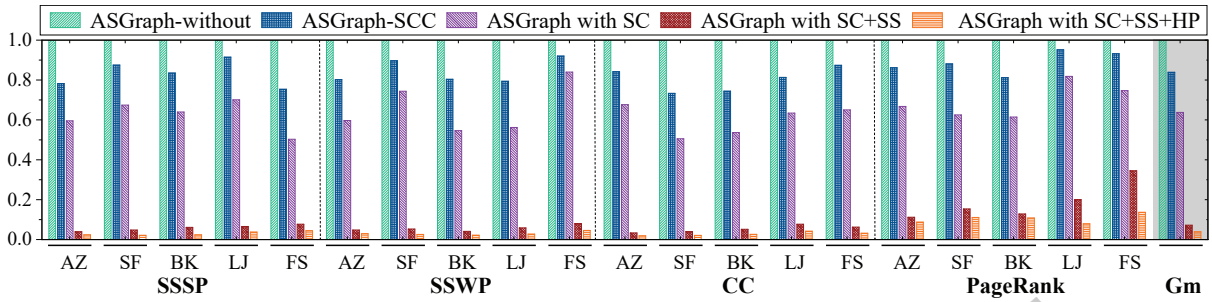


Fig. 14. Benefit breakdown of ASGraph

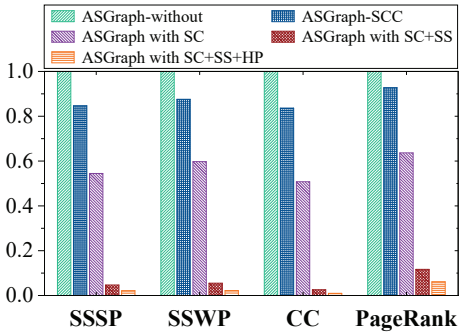


Fig. 15. Breakdown analyses on energy consumption over FS

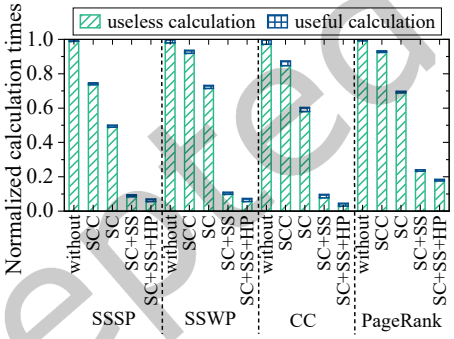


Fig. 16. Breakdown analyses on matrix calculation times over FS

native asynchronous execution model[4] without using our proposed optimization strategies and ASGraph-SCC is the version of ASGraph that only applies vertex reordering following the topological order of the SCCs. We can observe that the performance improvements mainly come from SC and SS, which is consistent with our expectations. Figure 14 shows that ASGraph-SCC achieves 1.1×~1.4× performance improvement compared to ASGraph-without. This is because ASGraph-SCC can concentrate the non-zero elements in the adjacency matrix and also enables the vertex states to be propagated along the topological order among the SCCs, achieving fewer redundant computations. When using the SC[5], it dynamically constructs the tightly connected subgraphs by tracking the dependencies between the active vertices and their successors. Thus, it can efficiently improve the crossbar density and enables only active vertices and their successors to loaded for processing. The SC is the basis for later optimizations and the finding of the tightly connected subgraphs enables us to propose strategies to efficiently schedule the processing of subgraphs. As shown in Figure 14, SC reduces execution time by 16%~49.7% compared to ASGraph-without. When using the SC and SS, the execution time can be reduced by 65.5%~96%. SS define a value for each subgraph and schedule the processing order of them so that the new vertex states of the subgraphs can be propagated along the dependencies between vertices as much as possible. When using the HP, it can further improve the performance by applying a hybrid processing scheme and reducing the redundant computation caused by the untimely state propagations for the processing of the subgraphs in each row. Figure 14 shows that the execution time can be finally reduced by 86.3%~97.9% when using SC, SS, and HP.

[4]There is no barrier between iterations, and the new states of each subgraph are allowed to be immediately used by other subgraphs. Like existing solutions [11, 44, 69], the graph is split into a series of subgraphs, which are handled by ASGraph-without in a round-robin way.
[5]ASGraph_with_SC dynamically constructs the subgraphs with good internal connectivity and then handles these subgraphs along their construction order. Note that the new states of each subgraph are also allowed to be immediately used by other subgraphs.

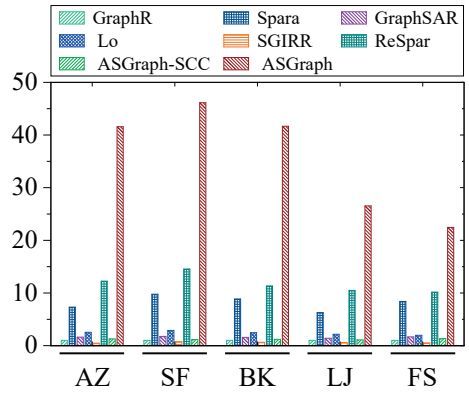


Fig. 17. Performance of various ReRAM-based accelerators normalized to that of GraphR

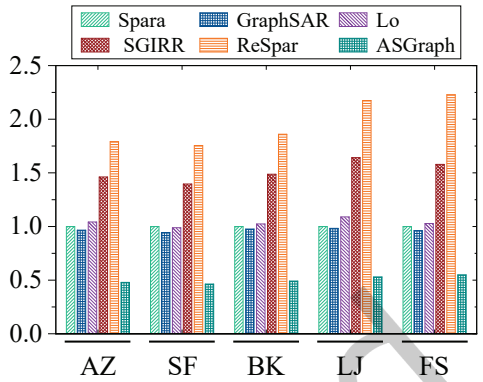
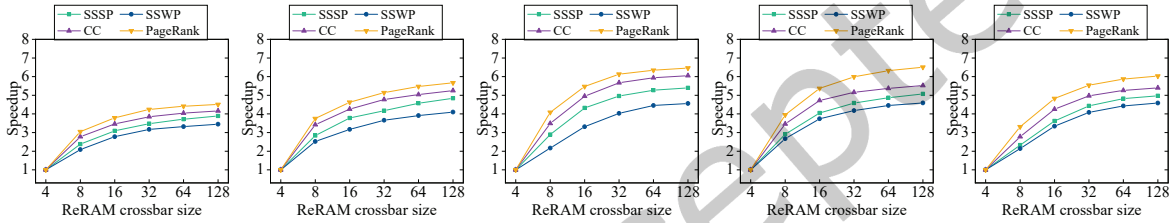


Fig. 18. Preprocessing overhead of different solutions normalized to that of Spara

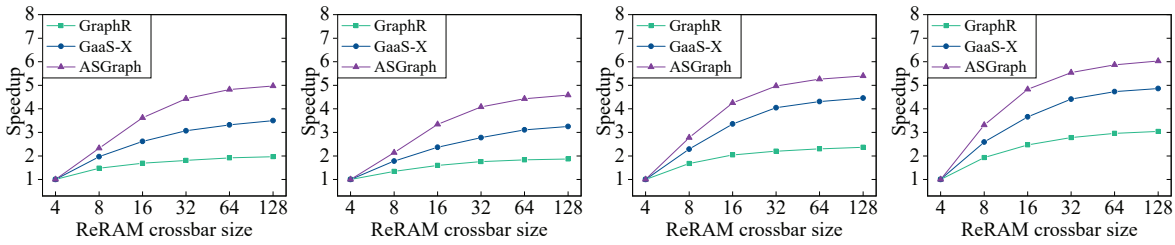


(a) AZ (b) SF (c) BK (d) LJ (e) FS

Fig. 19. Sensitivity to ReRAM crossbar size of ASGraph on different datasets

Figure 15 and Figure 16 conduct the breakdown analyses on energy, matrix calculation times, and useless updates of our proposed optimization strategies. These figures show that the reduction of useless calculations and energy savings also mainly come from SC and SS. In detail, SC reduces useless matrix calculation times by 37.8% and achieves 1.8× energy saving in comparison with ASGraph-without. When using SC and SS, the useless matrix calculation times can be reduced by 88.9% and the energy saving is increased to 18.9×. Finally, when using SC, SS, and HP, the useless matrix calculation times can be reduced by 93.6%, and the energy saving is increased to 43.6×.

Figure 17 evaluates the performance of the ReRAM-based accelerators with different vertex reordering methods (i.e., Spara [69], GraphSAR [11], Lo [26], SGIRR [47], ReSpar [18], ASGraph-SCC, and ASGraph) normalized to that of GraphR when running SSSP, which is a representative graph algorithm used in Graph 500 Benchmark [2]. From this figure, we can find that ASGraph-SCC achieves comparable performance to GraphSAR, Lo, and SGIRR, but worse than Spara and ReSpar. Note that, as shown in Figure 18, although ASGraph-SCC performs worse than Spara and ReSpar, it requires less preprocessing overhead in comparison with them. However, although the above vertex reordering solutions can increase the opportunities to skip redundant writes and computations with zero-valued elements in the adjacency matrix, they still suffer from significant redundant computations due to the irregular vertex state propagations. Compared with them, based on the vertex reordering following the topological order of the SCCs, ASGraph dynamically constructs the subgraph based on the dependencies between vertices in each SCC (i.e., SC) and then assigns the high-value subgraphs to be preferentially processed (i.e., SS), efficiently regularizing the state propagations of the ReRAM-based graph processing. Besides, ASGraph further employs a hybrid processing scheme (i.e., HP) to further accelerate the state propagations of the tightly connected subgraph. This way, as shown in Figure 17, ASGraph outperforms Spara, GraphSAR, Lo, SGIRR, ReSpar, and ASGraph-SCC by 4.3×, 21.5×, 14.4×, 58.8×, 2.9×, and 28.5× on average, respectively.



(a) SSSP (b) SSWP (c) CC (d) PageRank

Fig. 20. Sensitivity to ReRAM crossbar size of different solutions over FS

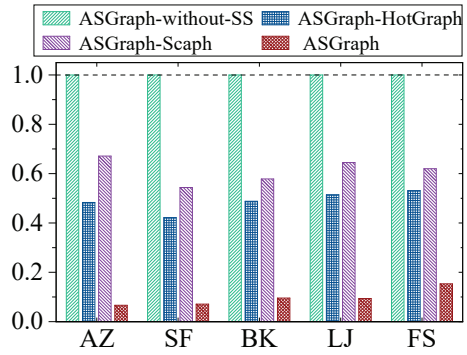


Fig. 21. Normalized execution time when using different scheduling methods

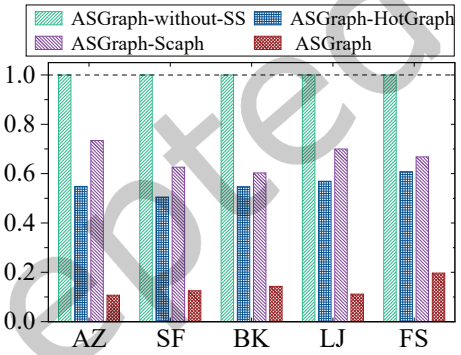


Fig. 22. Normalized energy consumption when using different scheduling methods

4.4 Sensitivity Study

Figure 19 have evaluated the performance of ASGraph with different ReRAM crossbar sizes, where the crossbar size ranges from 4×4 to 128×128. It shows that better performance can be obtained by ASGraph when the ReRAM crossbar size becomes larger. Specifically, when the ReRAM crossbar size is less than 8×8, 8×8, 16×16, 16×16, and 32×32 for AZ, SF, BK, LJ, and FS, respectively, the performance improvement of ASGraph gains dramatically. This is because the averaged edges of each vertex (i.e., $\#Edges/\#Vertices$, where $\#Edges$ and $\#Vertices$ represent the number of edges and vertices in the graph) in AZ, SF, BK, LJ, and FS are 5, 8, 11, 14, and 28, respectively. These are far less than the crossbar size used (e.g., 128×128) and also distributed in different crossbars with more zeros in each row. Thus, when the crossbar size is less than the averaged edges per vertex (e.g., $<$ 32×32 for FS), ASGraph improves the performance dramatically. If the crossbar size is greater than 32×32 for FS, the speedup growth gradually becomes slow. To strike a balance between the ReRAM crossbar size and performance, the better crossbar size can be set as 8×8, 8×8, 16×16, 16×16, and 32×32 for AZ, SF, BK, LJ, and FS, respectively. As depicted in Figure 20, ASGraph always achieves better performance than existing solutions under different ReRAM crossbar sizes.

Figure 21 evaluates the effectiveness of different scheduling methods, where ASGraph-without-SS is the version of ASGraph that disabled the *Value-Driven Subgraph Scheduling* (SS) strategy, and ASGraph-HotGraph and ASGraph-Scaph are the versions of ASGraph-without-SS that apply the scheduling methods in HotGraph [59] and Scaph [68], respectively. The results show that the value-driven subgraph scheduling strategy of ASGraph always outperforms the scheduling strategies of ASGraph-HotGraph and ASGraph-Scaph. This is because ASGraph accurately assigns higher priority to subgraphs that have accumulated significant state propagations

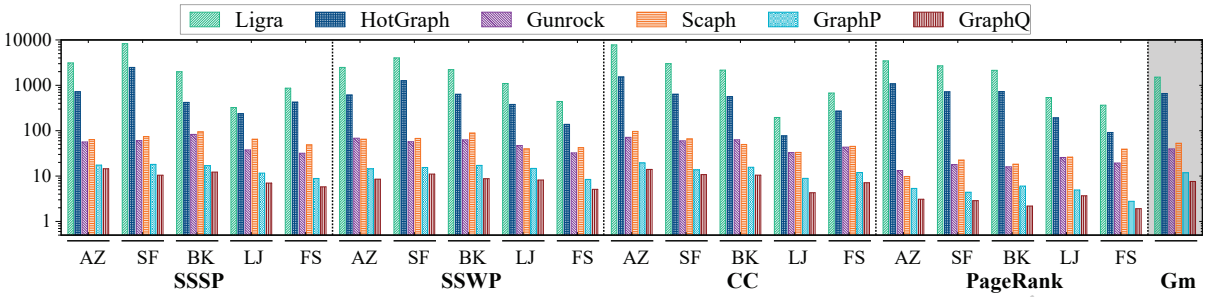


Fig. 23. Speedup compared to CPU, GPU, and PIM platforms

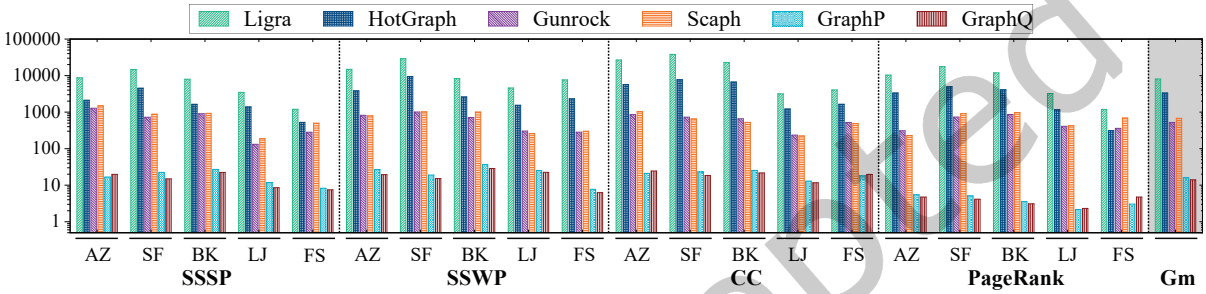


Fig. 24. Energy saving compared to CPU, GPU, and PIM platforms

from their neighbors, achieving more regularized state propagations. Figure 22 shows that ASGraph requires less energy consumption compared with other solutions.

4.5 Comparison to Other Platforms

The performance comparison of ASGraph with CPU platform is shown in Figure 23. We run Ligra [43] and HotGraph [59] on a 64-core Intel Xeon CPU and normalize the performance of ASGraph to them. The experimental result shows that the geometric mean of speedups of ASGraph compared to Ligra and HotGraph are 1514.9× and 661.1×, respectively. Compared to CPU platform, ASGraph performs better on AZ, SF, and BK datasets and achieves lower speedup on LJ and FS datasets. This is because the adjacency matrices of LJ and FS datasets are more sparse compared to other datasets, so the effective computation accounts for a smaller proportion when implementing graph algorithms using SpMV operation. Note that we use $\#Edges/(\#Vertices)^2$ to represent the density of a graph, and with the density decreases, the sparsity increases. For AZ, SF, BK, LJ, and FS, their values of $\#Edges/(\#Vertices)^2$ are $1.8e^{-5}$, $2.9e^{-5}$, $1.6e^{-5}$, $2.9e^{-6}$ and $4.2e^{-7}$, respectively. Figure 24 shows the energy saving of ASGraph normalized to CPU platform. ASGraph achieves 8064.2× and 3347.8× energy saving compared to Ligra and HotGraph, respectively.

We run Gunrock [48] and Scaph [68] on a GPU platform with 6912 CUDA cores and normalize the performance of ASGraph to them. As shown in Figure 23, the geometric mean of speedups of ASGraph compared to Gunrock and Scaph are 39.6× and 52.8×, respectively. The energy saving of ASGraph normalized to GPU platform is shown in Figure 24. It shows that ASGraph achieves 521.9× and 676.5× energy savings compared to Gunrock and Scaph, respectively.

We compared the performance of ASGraph with two cutting-edge *processing-inmemory* (PIM) based graph processing accelerators, i.e., GraphP [57] and GraphQ [72], which use the same configurations in [72]. Figure 23 and Figure 24 show that ASGraph outperforms GraphP and GraphQ by 2.7×~19.5×, 1.9×~14.1× and achieves 2.1×~37.1×, 2.3×~28.5× energy savings, respectively.

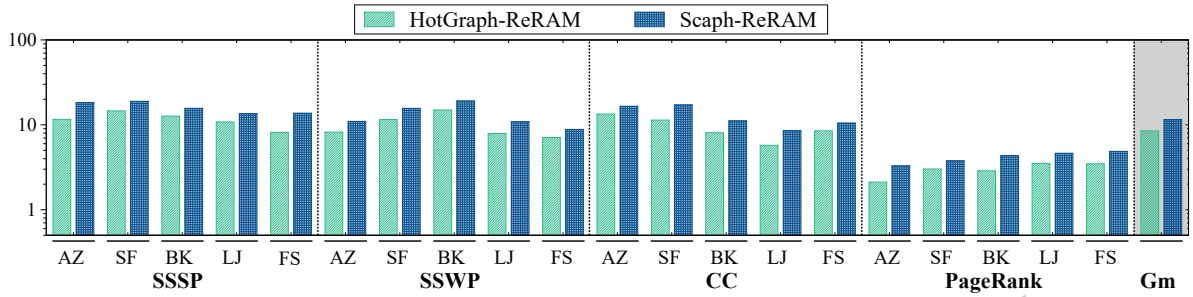


Fig. 25. Speedup compared to HotGraph-ReRAM and Scaph-ReRAM

4.6 Comparison to Different Asynchronous Graph Processing Solutions

To evaluate the impact of the asynchronous graph processing solutions, we have implemented the approaches of HotGraph [59] and Scaph [68] on the ReRAM architectures to obtain ReRAM-based accelerators HotGraph-ReRAM and Scaph-ReRAM, respectively. Specifically, HotGraph-ReRAM extracts a backbone structure from the original graph and then prioritizes the partitions of this backbone structure for processing over the ReRAM crossbar. In Scaph-ReRAM, the partitions are categorized into either high-value or low-value partitions. After that, the high-value partitions are fully loaded and iterated over repeatedly on the ReRAM crossbar, while only the active graph data in the low-value partitions is loaded and handled over the ReRAM crossbar. Note that the ReRAM crossbar size and the number of ReRAM crossbars in HotGraph-ReRAM and Scaph-ReRAM are the same as those in ASGraph. Figure 25 evaluates the performance of ASGraph in comparison with HotGraph-ReRAM and Scaph-ReRAM. This results show that ASGraph outperforms HotGraph-ReRAM and Scaph-ReRAM by 2.1×~14.9× and 3.2×~19.1×, respectively.

5 Related Work

Many software systems [15, 20, 22, 39, 54, 65, 67, 71] are developed based on CPU or GPU platforms to achieve efficient graph processing. Ligra [43] is a lightweight graph processing system toward the shared-memory architecture. GraphX [16] is a distributed graph processing system that takes advantage of distributed dataflow system to realize the low-cost fault tolerance in graph processing. Some GPU-based graph processing systems [48, 51, 60] utilize the massive CUDA cores and high memory bandwidth in GPU to achieve efficient processing. However, the effectiveness of these software solutions over the conventional architecture faces restrictions due to the intrinsic random access and the large amount of data movement of graph processing applications. To overcome these limitations, some ASIC-based accelerators [7, 17, 28, 35–37, 50, 55, 64, 66] are further proposed to accelerate graph processing using specific memory organizations and custom execution pipelines. Graphicionado [17] is the first domain-specific graph accelerator that can reduce random memory accesses. Graphpulse [36] uses an event-driven execution model to reduce synchronization overhead and optimize memory access patterns. Nevertheless, these studies still suffer from substantial data movements when serving graph processing applications.

By integrating computing logic into memory, *processing-in-memory* (PIM) techniques can tackle the "memory wall" challenge to enable high-performance graph processing. Tesseract [3] is the first PIM-based graph accelerator, which instantiates the custom computing units near the DRAM arrays to efficiently perform parallel graph operations through utilizing their large internal bandwidth. GraphP [57] adopts a hierarchical communication approach to achieve lower communication and energy consumption. GraphQ [72] proposes a hybrid execution model that essentially eliminates irregular data movement. However, these solutions still suffer from the redundant computation overhead caused by irregular state propagations. GraphR [44] proposes the first ReRAM-based graph processing accelerator by mapping the SpMV-based graph operations to crossbars. To solve the sparsity problem

in ReRAM-based solutions, several graph reordering methods are further proposed. GraphSAR [11], Lo [26], and SGIRR [47] employ the specific graph clustering methods to aggregate non-zero elements in the adjacency matrix to few crossbar arrays, alleviating the processing of more crossbar-sized sub-matrices with full-zero elements. To address both crossbar sparsity and activation sparsity, Spara [69] not only reorders the source and destination vertices of a graph, but also removes the processing of many inactive edges via an in-situ merging approach. However, the effectiveness of the above reordering methods is constrained by the original ordering of matrix rows. To overcome this limitation, ReSpar [18] proposes a matrix reordering method to aggregate matrix rows with similar non-zero column elements together into clusters and concentrate non-zero column elements to create more full-zero crossbar arrays. For better performance, GaaS-X [5] performs calculations directly using the sparse data representation, reducing the sparse to dense conversion overhead and redundant computation overhead on invalid edges. Nevertheless, they still suffer from significant redundant computation overhead due to the irregular processing of subgraphs. In comparison, ASGraph dynamically constructs the subgraphs based on dependencies between vertices, regularizes the state propagations, and applies a hybrid processing scheme to reduce computation overhead and achieve better performance.

Many graph learning accelerators [9, 24, 41, 52] have been proposed that leverage PIM techniques. gPIM [19] offloads memory-bound aggregation and combination to the PIM enabled architecture while preserving GPU to perform compute-intensive combination. GCN [32] proposes multiple engines incorporating a node-stationary dataflow to reduce the off-chip memory accesses and increase the reuse of node feature data in GCN inference. TARe [53] is proposed to efficiently support both weight-static and data-static execution modes for graph learning via a task-adaptive in-situ computing method, achieving higher processing throughput and less data movement in total. Although these solutions can efficiently map the SpMV operations over the ReRAM crossbar, they also suffer from significant redundant computation overhead to the same graph data due to the irregular state propagations.

To enhance graph processing performance, some asynchronous graph processing techniques [61, 63] have been proposed. HotGraph [59] extracts a backbone structure from the original graph to reduce cross-partition state propagation during asynchronous graph processing. Scaph [68] designs heterogeneous graph engines to differentially schedule and process partitions. However, when these solutions are applied to ReRAM-based architectures, they still suffer from redundant computations due to the irregular state propagation among the matrix-formatted subgraphs. Compared with them, ASGraph can efficiently regularize the state propagations among the subgraphs and also achieve fast state propagations of the constructed tightly connected subgraph, significantly reducing the redundant computation of ReRAM-based graph processing.

6 Conclusion

This paper proposes a dependency-aware ReRAM-based accelerator *ASGraph* for efficient asynchronous graph processing. ASGraph constructs the subgraph based on dependencies between vertices' states, regularizes the state propagations according to the values of subgraphs, and applies a hybrid processing scheme to minimize the redundant computation overhead. In this way, ASGraph can greatly reduce the amount of computation on the crossbars, speeding up the convergence of graph algorithms and achieving better performance. The experimental result shows that ASGraph achieves 25.5× and 4.8× speedup and 70.8× and 2.2× energy saving compared with the state-of-the art ReRAM-based graph processing accelerators GraphR [44] and GaaS-X [5], respectively.

References

[1] [n. d.]. CACTI. http://www.hpl.hp.com/research/cacti/
[2] [n. d.]. Graph 500 Benchmark. https://graph500.org/
[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 105–117.

[4] Chris Biemann. 2006. Chinese whispers-an efficient graph clustering algorithm and its application to natural language processing problems. In *Proceedings of TextGraphs: the first workshop on graph based methods for natural language processing*. 73–80.
[5] Nagadastagiri Challapalle, Sahithi Rampalli, Linghao Song, Nandhini Chandramoorthy, Karthik Swaminathan, John Sampson, Yiran Chen, and Vijaykrishnan Narayanan. 2020. GaaS-X: Graph analytics accelerator supporting sparse data representation using crossbar architectures. In *Proceedings of the 2020 47th Annual International Symposium on Computer Architecture*. 433–445.
[6] Runzhe Chen, Guandong Lu, Yakai Wang, Rui Zhang, Zheng Hu, Yanming Miao, Zhifang Cai, Jingwen Leng, and Minyi Guo. 2025. BAFT: bubble-aware fault-tolerant framework for distributed DNN training with hybrid parallelism. *Frontiers of Computer Science* 19, 1 (2025), 191102.
[7] Xinyu Chen, Yao Chen, Feng Cheng, Hongshi Tan, Bingsheng He, and Weng-Fai Wong. 2022. ReGraph: Scaling Graph Processing on HBM-enabled FPGAs with Heterogeneous Pipelines. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture*. 1342–1358.
[8] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*. 85–98.
[9] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture*. 27–39.
[10] Simon H Corston, William B Dolan, Lucy H Vanderwende, and Lisa Braden-Harder. 2005. System for processing textual inputs using natural language processing techniques. US Patent 6,901,399.
[11] Guohao Dai, Tianhao Huang, Yu Wang, Huazhong Yang, and John Wawrzynek. 2019. GraphSAR: A sparsity-aware processing-in-memory architecture for large-scale graph processing on ReRAMs. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 120–126.
[12] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing* 8, 1 (2021), 1–70.
[13] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P Jouppi. 2012. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 7 (2012), 994–1007.
[14] Brendan J Frey. 1998. *Graphical models for machine learning and digital communication*.
[15] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. 17–30.
[16] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX symposium on operating systems design and implementation*. 599–613.
[17] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. 56:1–56:13.
[18] Yi-Jou Hsiao, Chin-Fu Nien, and Hsiang-Yun Cheng. 2021. ReSpar: Reordering Algorithm for ReRAM-based Sparse Matrix-Vector Multiplication Accelerator. In *Proceedings of the 39th IEEE International Conference on Computer Design, ICCD 2021*. 260–268.
[19] Hai Jin, Dan Chen, Long Zheng, Yu Huang, Pengcheng Yao, Jin Zhao, Xiaofei Liao, and Wenbin Jiang. 2023. Accelerating Graph Convolutional Networks Through a PIM-Accelerated Approach. *IEEE Trans. Comput.* 72, 9 (2023), 2628–2640.
[20] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 31–46.
[21] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection.
[22] Xiaofei Liao, Jin Zhao, Yu Zhang, Bingsheng He, Ligang He, Hai Jin, and Lin Gu. 2022. A Structure-Aware Storage Optimization for Out-of-Core Concurrent Graph Processing. *IEEE Trans. Comput.* 71, 7 (2022), 1612–1625.
[23] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing* 7, 1 (2003), 76–80.
[24] Jingyu Liu, Shi Chen, and Li Shen. 2025. A comprehensive survey on graph neural network accelerators. *Frontiers of Computer Science* 19, 2 (2025), 192104.
[25] Xianzhu Liu, Zhijian Ji, and Ting Hou. 2019. Graph partitions and the controllability of directed signed networks. *SCIENCE CHINA Information Sciences* 62, 4 (2019), 42202:1–42202:11.
[26] Ting-Shan Lo, Chun-Feng Wu, Yuan-Hao Chang, Tei-Wei Kuo, and Wei-Chen Wang. 2021. Space-efficient Graph Data Placement to Save Energy of ReRAM Crossbar. In *Proceedings of the 2021 IEEE/ACM International Symposium on Low Power Electronics and Design*. 1–6.
[27] Rada Mihalcea and Dragomir Radev. 2011. *Graph-based natural language processing and information retrieval*.
[28] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on*

Microarchitecture. 1–14.
[29] LMU Munich. 2021. CPU Energy Meter. https://github.com/sosy-lab/cpu-energy-meter
[30] Boris Murmann. [n. d.]. ADC Performance Survey 1997-2024. [Online]. Available: https://github.com/bmurmann/ADC-survey.
[31] Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*.
[32] Challapalle Nagadastagiri, Swaminathan Karthik, Chandramoorthy Nandhini, and Narayanan Vijaykrishnan. 2021. Crossbar based processing in memory accelerator architecture for graph convolutional networks. In *Proceedings of the 2021 IEEE/ACM International Conference On Computer Aided Design*. 1–9.
[33] Dimin Niu, Cong Xu, Naveen Muralimanohar, Norman P Jouppi, and Yuan Xie. 2013. Design of cross-point metal-oxide ReRAM emphasizing reliability and cost. In *Proceedings of the 2013 IEEE/ACM International Conference on Computer-Aided Design*. 17–23.
[34] NVIDIA. 2023. Nvidia system management interface. https://developer.nvidia.com/nvidia-system-management-interface
[35] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven M. Burns, and Özcan Özturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture*. 166–177.
[36] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2020. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. 908–921.
[37] Shafiur Rahman, Mahbod Afarin, Nael Abu-Ghazaleh, and Rajiv Gupta. 2021. Jetstream: Graph analytics on streaming data with event-driven hardware accelerator. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1091–1105.
[38] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, Rolf Risen, Jeanine Cook, Paul Rosenfeld, Elliot Cooper-Balis, and Bruce Jacob. 2011. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review* 38, 4 (2011), 37–42.
[39] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 472–488.
[40] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. 2007. Collaborative filtering recommender systems. In *The adaptive web: methods and strategies of web personalization*. 291–324.
[41] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture*. 14–26.
[42] Xinyang Shen, Xiaofei Liao, Long Zheng, Yu Huang, Dan Chen, and Hai Jin. 2024. ARCHER: a ReRAM-based accelerator for compressed recommendation systems. *Frontiers of Computer Science* 18, 5, Article 185607 (2024), 185607 pages.
[43] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.
[44] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture*. 531–543.
[45] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.
[46] Frank Edward Walter, Stefano Battiston, and Frank Schweitzer. 2008. A model of a trust-based recommendation system on a social network. *Autonomous Agents and Multi-Agent Systems* 16 (2008), 57–74.
[47] Cheng-Yuan Wang, Yao-Wen Chang, and Yuan-Hao Chang. 2022. SGIRR: Sparse Graph Index Remapping for ReRAM Crossbar Operation Unit and Power Optimization. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 165:1–165:7.
[48] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.
[49] H-S Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming-Jinn Tsai. 2012. Metal–oxide RRAM. *Proc. IEEE* 100, 6 (2012), 1951–1970.
[50] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2019. Alleviating Irregularity in Graph Analytics Acceleration: a Hardware/Software Co-Design Approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 615–628.
[51] Carl Yang, Aydın Buluç, and John D Owens. 2022. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Trans. Math. Software* 48, 1 (2022), 1–51.
[52] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I-Ching Tseng, Han-Wen Hu, Hung-Sheng Chang, and Hsiang-Pang Li. 2019. Sparse ReRAM engine: joint exploration of activation and weight sparsity in compressed neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*. 236–249.
[53] He Yintao, Wang Ying, Liu Cheng, Li Huawei, and Li Xiaowei. 2021. Tare: task-adaptive in-situ reram computing for graph learning. In *Proceedings of the 58th ACM/IEEE Design Automation Conference*. 577–582.
[54] Pingpeng Yuan, Wenya Zhang, Changfeng Xie, Hai Jin, Ling Liu, and Kisung Lee. 2014. Fast Iterative Graph Computation: A Path Centric Approach. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis*.

401–412.
[55] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. *ACM SIGPLAN Notices* 53, 2 (2018), 593–607.
[56] Jianpeng Zhang, Hongchang Chen, Dingjiu Yu, Yulong Pei, and Yingjun Deng. 2023. Cluster-preserving sampling algorithm for large-scale graphs. *SCIENCE CHINA Information Sciences* 66, 1 (2023).
[57] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture*. 544–557.
[58] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2013. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Transactions on Parallel and Distributed Systems* 25, 8 (2013), 2091–2100.
[59] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Guang Tan, and Bing Bing Zhou. 2017. HotGraph: Efficient Asynchronous Processing for Real-World Graphs. *IEEE Trans. Comput.* 66, 5 (2017), 799–809.
[60] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. 2019. DiGraph: An Efficient Path-based Iterative Directed Graph Processing System on Multiple GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 601–614.
[61] Yu Zhang, Xiaofei Liao, Hai Jin, Ligang He, Bingsheng He, Haikun Liu, and Lin Gu. 2021. DepGraph: A Dependency-Driven Accelerator for Efficient Iterative Graph Processing. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture*. 371–384.
[62] Yu Zhang, Xiaofei Liao, Xiang Shi, Hai Jin, and Bingsheng He. 2017. Efficient disk-based directed graph processing: A strongly connected component approach. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (2017), 830–842.
[63] Jin Zhao, Yun Yang, Yu Zhang, Xiaofei Liao, Lin Gu, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, Xinyu Jiang, and Hui Yu. 2022. TDGraph: a topology-driven accelerator for high-performance streaming graph processing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 116–129.
[64] Jin Zhao, Yu Zhang, Jian Cheng, Yiyang Wu, Chuyue Ye, Hui Yu, Zhiying Huang, Hai Jin, Xiaofei Liao, Lin Gu, and Haikun Liu. 2023. SaGraph: A Similarity-aware Hardware Accelerator for Temporal Graph Processing. In *Proceedings of the 60th ACM/IEEE Design Automation Conference, DAC 2023, San Francisco*. 1–6.
[65] Jin Zhao, Yu Zhang, Ligang He, Qikun Li, Xiang Zhang, Xinyu Jiang, Hui Yu, Xiaofei Liao, Hai Jin, Lin Gu, Haikun Liu, Bingsheng He, Ji Zhang, Xianzheng Song, Lin Wang, and Jun Zhou. 2023. GraphTune: An Efficient Dependency-Aware Substrate to Alleviate Irregularity in Concurrent Graph Processing. *ACM Transactions on Architecture and Code Optimization* 20, 3 (2023), 37:1–37:24.
[66] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, and Haikun Liu. 2021. LCCG: a locality-centric hardware accelerator for high throughput of concurrent graph processing. In *Proceedings of the 2021 International Conference for High Performance Computing, Networking, Storage and Analysis*. 45.
[67] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. 2019. GraphM: an efficient storage system for high throughput of concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 3:1–3:14.
[68] Long Zheng, Xianliang Li, Yaohui Zheng, Yu Huang, Xiaofei Liao, Hai Jin, Jingling Xue, Zhiyuan Shao, and Qiang-Sheng Hua. 2020. Scaph: Scalable GPU-Accelerated Graph Processing with Value-Driven Differential Scheduling. In *Proceedings of the 2020 USENIX Annual Technical Conference*. 573–588.
[69] Long Zheng, Jieshan Zhao, Yu Huang, Qinggang Wang, Zhen Zeng, Jingling Xue, Xiaofei Liao, and Hai Jin. 2020. Spara: An energy-efficient ReRAM-based accelerator for sparse graph analytics applications. In *Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium*. 696–707.
[70] Minxuan Zhou, Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. GRAM: graph processing in a ReRAM-based computational memory. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 591–596.
[71] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference*. 375–386.
[72] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. Graphq: Scalable pim-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 712–725.

Received 27 December 2023; revised 3 July 2024; accepted 30 July 2024