GraphSAR: A Sparsity-Aware Processing-in-Memory Architecture for Large-scale Graph Processing on ReRAMs

Guohao Dai*, Tianhao Huang**, Yu Wang*, Huazhong Yang*, John Wawrzynek***
*Dept. of EE, BNRist, Tsinghua University, **Massachusetts Institute of Technology, ***University of California, Berkeley,
*{dgh14@mails.,yu-wang@,yanghz@}tsinghua.edu.cn,**tianhaoh@mit.edu,***johnw@eecs.berkeley.edu

Abstract

Large-scale graph processing has drawn great attention in recent years. The emerging metal-oxide resistive random access memory (ReRAM) and ReRAM crossbars have shown huge potential in accelerating graph processing. However, the sparse feature of natural graphs hinders the performance of graph processing on ReRAMs. Previous work of graph processing on ReRAMs stored and computed edges separately, leading to high energy consumption and long latency of transferring data. In this paper, we present GraphSAR, a **s**parsity-**a**ware processing-in-memory large-scale graph processing accelerator on **R**eRAMs. Computations over edges are performed in the memory, eliminating overheads of transferring edges. Moreover, graphs are divided considering the sparsity. Subgraphs with low densities are further divided into smaller ones to minimize the waste of memory space. According to our extensive experimental results, GraphSAR achieves 4.43x energy reduction and 1.85x speedup (8.19x lower energy-delay product, EDP) against previous graph processing architecture on ReRAMs (GraphR [1]).

1 Introduction

Graphs are widely used to model both data and relationships in real-world problems, including neural network modeling, social network analysis, etc. The demand for large-scale graph processing has skyrocketed in recent years. Many graph processing systems have been put forward, including single PC-based [2–5], cluster-based [6, 7], hardware specialized accelerators [8–11], and processing-in-memory designs [12–14].

The emerging metal-oxide resistive random access memory (ReRAM) and ReRAM crossbars show huge potential in energy efficient processing-in-memory operations [15, 16], especially for matrix-vector multiplications [17]. On the other hand, graph algorithms can be naturally represented by updating destination vertices using source vertices, through the adjacency matrix. Many graph processing architectures/accelerators based on ReRAMs have been proposed (e.g., RPBFS [18], GraphR [1], and HyVE [19]), achieving great speedup and energy efficiency improvement.

The sparse natural graphs are hard to be stored into ReRAM crossbars in the form of adjacency matrix. Instead, in the design of RPBFS, GraphR, and HyVE, edges in the graph are stored using the edge list format. RPBFS adopts a shared memory to store values of vertices. GraphR converts edge list into adjacency matrix, and then stores the matrix into the ReRAM crossbar. HyVE further designs memory hierarchy for vertex storage, which is not mentioned in GraphR. Then, HyVE uses conventional CMOS circuits to process the subgraph edge by edge. Although using ReRAM crossbars can process multiple edges in just one operation, it still suffers from two main problems. (1) **Heavy writing overheads.** Previous work like GraphR [1] converted the edge list into the adjacency matrix by sequentially writing each edge into the ReRAM crossbar. Writing data to ReRAMs leads to higher energy consumption and longer latency compared with reading/computing over ReRAMs, especially when multi-level cells are required. (2) **No parallelism inside a block.** In GraphR, edges in a subgraph are not processed in parallel because edges are sequentially written to ReRAM crossbars.

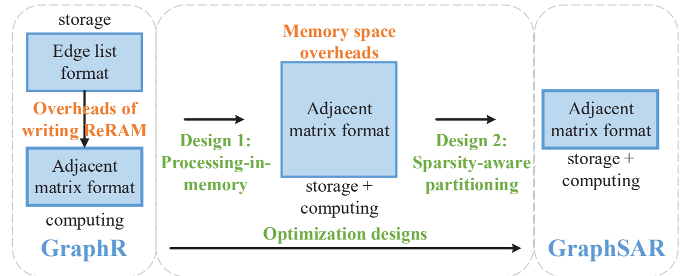

Figure 1: Comparison between GraphR and GraphSAR, including two optimization designs in GraphSAR.

Processing-in-memory is a promising solution to overcoming heavy overheads of writing data to ReRAM crossbars. However, directly storing all subgraphs using the adjacency matrix leads to heavy memory overheads. Thus, we present GraphSAR, a sparsity-aware processing-in-memory graph processing accelerator on ReRAMs. Computations are performed directly in the memory. Subgraphs with low densities are divided into smaller ones to reduce memory space waste. The detailed contributions are summarized as follows:

- **Processing-in-memory graph processing.** GraphSAR directly processes edges in the memory where edges are stored. In this way. GraphSAR avoids high energy consumption and latency of writing edge values to computing units.
- **Sparsity-aware graph processing.** Subgraphs with low densities are divided into smaller ones in order to achieve memory space efficiency. Compared with storing all subgraphs using a large adjacency matrix (8×8), we reduce the memory space requirements from 45.87x to 1.54x (normalized to the non processing-in-memory implementation).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPDAC '19, January 21–24, 2019, Tokyo, Japan
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6007-4/19/01…$15.00
https://doi.org/10.1145/3287624.3287637

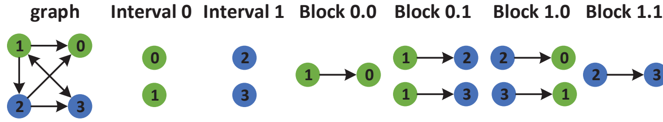

Figure 2: Interval-block graph partitioning, 4 vertices and 6 edges are divided into 2 intervals and 2 × 2 = 4 blocks.

- **Fewer subgraphs and lower cell bits.** We propose a lightweight graph clustering method to reduce processed subgraphs by up to 70.38%, reducing the energy-efficient product (EDP) by 1.78x. We also show that only single-bit cells are required for unweighted graph algorithms, leading to 3.77x lower EDP.

We take the design of GraphR [1] as the baseline and compare it with GraphSAR in Figure 1. Experimental results show that, GraphSAR can achieve 4.43x energy reduction and 1.85x speedup (8.19x lower EDP) against GraphR, with only 1.54x memory space.

2 Background and Related Work

2.1 Graph Processing and Partitioning Model

The Gather-Apply-Scatter (GAS) model is used to describe different graph algorithms. In GAS, processing one vertex includes: (1) Gathering values from incoming neighbors; (2) Applying gathered values to get a new value; (3) Scattering the new value to all outgoing neighbors. Different algorithms only differ in how to compute the new value. There are two main implementations of GAS model:

- Vertex-centric: Iterating over vertices. A vertex's value is propagated to all neighbors, and neighbors are updated.
- Edge-centric: Iterating over edges. The value of an edge's source vertex is propagated to updating the destination vertex.

The edge-centric model is first proposed by X-stream [3] to ensure the locality of graph processing. The interval-block graph partitioning method is widely used in previous systems [4, 5]. Vertices are divided into disjointed intervals, and edges are divided into corresponding blocks. Edges in a block are from vertices in an interval to vertices in another interval, shown in Figure 2. By adopting both edge-centric model and interval-block partitioning method, graph algorithms can be executed block by block. Within a block, each edge is processed to propagate the value from the source vertex to the destination vertex. Algorithm 1 shows the flow of graph processing under the edge-centric model after partitioning.

2.2 Graph Processing on ReRAMs

The low read latency and high energy efficiency make ReRAMs the promising solution as edge storage in graph processing. Previous ReRAM-based graph processing systems all store edges in ReRAMs to perform the edge-centric model. RPBFS [18] uses a shared memory to store values of all vertices. Such implementation leads to the scalability problem. The size of shared memory increases drastically when graphs become larger or algorithm changes (e.g, when running PageRank, the size of each vertex increases by at least 32x (from 1-bit `boolean` to 32-bit `float`)). From the experimental results of RPBFS we can see, when the size of crossbar increases, larger graphs achieve more improvements against smaller graphs. This means that, the performance of RPBFS will decrease drastically when graphs become larger compared with the fixed-size crossbar.

Algorithm 1 The edge-centric model after the interval-block partitioning

Input: $G = (V, E)$, initialization
Output: Updated V

```
1: Initialization()
2: while not finished do
3:     for each e_{i.j} ∈ each B_{x.y} do
4:         value(v_i) = Update(v_i, v_j, e_{i,j})
5:     end for
6: end while
7: return V
```

To ensure the scalability, GraphR [1] and HyVE [19] divide a graph into subgraphs using interval-block partitioning. Only a fixed-size subgraph is processed without accessing other parts of the graph. GraphR writes a subgraph into the ReRAM crossbar in the form of adjacency matrix, and values of source/destination vertices are stored in registers. Then, GraphR performs graph operations using the crossbar. Edges are sequentially written to ReRAM crossbars to be converted into the adjacency matrix, leading to high energy consumption and long latency, especially when data are stored in multi-level cells of ReRAMs. We can define the effective parallelism of processing edges in a subgraph in Equation (1) by normalizing the total processing time to the time of processing one edge ($T_{write} + T_{read}$).

$$p_{effective} = \frac{T_{write} + T_{read}}{N_{edges} \times T_{write} + T_{read}} \times N_{edges} \tag{1}$$

In Equation (1), T_{write} and T_{read} represent latency of writing data to and performing computation on ReRAM crossbars, and N_{edges} represents the number of edges in a subgraph. We can see that $P_{effective}$ is always less than N_{edges}. Moreover, because of heavy overheads of 'Write-and-verify' operations [20], T_{write} is usually larger than T_{read} in ReRAM crossbars, thus $p_{effective}$ is around 1. $p_{effective}$ is linear to N_{edges} only when $\frac{T_{write}}{T_{read}} << 1$, which is unpractical in real devices. Thus, GraphR does not process edges in a subgraph in parallel. HyVE focuses on the memory hierarchy for vertices in graph processing. Edges are processed using the CMOS circuit one by one in HyVE. Thus, HyVE cannot process multiple edges in a subgraph in one operation.

3 GraphSAR Model

Writing adjacency matrices to crossbars leads to heavy overheads on energy consumption and latency. Storing the graph in the form of the adjacency matrix, rather than converting the graph from other formats into the adjacency matrix, is the promising solution to achieve the energy efficient graph processing on ReRAMs. However, due to the sparsity of graphs, directly storing a whole graph in the form of the adjacency matrix suffers from memory space inefficiency. Thus, in this section, we detail our hybrid-centric processing model and sparsity-aware graph partitioning scheme.

3.1 Hybrid-centric Model

Although GraphR can benefit from processing multiple edges in a block, the parallelism does not apply to blocks with only one edge, and suffers from blocks with low densities. Thus, we can directly process the only edge in blocks with one edge, and divide a larger

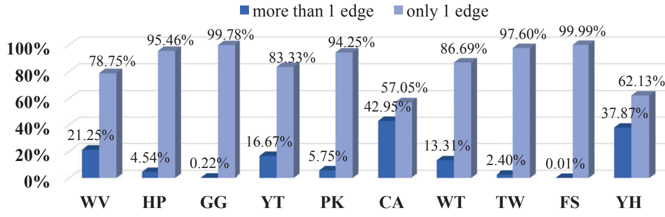

Figure 3: Percentage of subgraphs (8 source and 8 destination vertices) with only one edge in all subgraphs to be processed, graphs are from TABLE 1.

sparse blocks into several smaller ones. Note that when dividing the whole graph into small blocks (e.g., 8×8), these blocks also sparsely spread over the graph as edges. Thus, we can process these blocks using a block-centric model which is similar to the edge-centric model mentioned in Section 2.1. We define this block-centric model and further propose the hybrid-centric model as:

- Block-centric: Iterating over blocks. When processing a block, values of source vertices in the block is propagated through edges, and destination vertices are updated.
- Hybrid-centric: Blocks with only one edge are stored in the edge list (blocks with multiple edges are stored in the block list). Then, iterations are performed over edges and blocks, using edge/block-centric models correspondingly.

GraphSAR adopts this hybrid-centric model to process blocks with multiple edges and one edge separately. Based on this hybrid-centric model, we will detail the sparsity-aware partitioning scheme in the following section.

3.2 Sparsity-aware Graph Partitioning

In order to process edges in the form of the adjacency matrix, we can store all non-empty blocks using adjacency matrix format. However, as we can see in Figure 3, most non-empty blocks are sparse (only containing 1 edge). Thus, if we store all non-empty 8×8 blocks using adjacency matrix format, a huge amount of memory space is wasted. We normalize the memory space usage of directly storing all blocks into the adjacency matrix to storing all edges using the edge list in Figure 4. As we can see, 46.87x memory space is required on average, compared with using the edge list.

According to the hybrid-centric model, graphs can be stored in an edge list (edges in blocks with only one edge) and a block list (blocks with multiple edges) separately. Thus, we can save the memory space by storing all single edges using the edge list instead of an 8×8 adjacency matrix. From Figure 4 we can see, such method can reduce the required memory space to 5.77x on average (normalized to storing all edges using the edge list). However, in some graphs (e.g., WV and CA), the memory space requirement is still larger than 10x.

Such an idea inspires us that we do not need to store sparse blocks using a large adjacent matrix. We divide graphs into blocks of size 8×8 first[1]. Then, for blocks with only one edge, we store these edges into the edge list. We only store blocks with densities larger than a given threshold (we choose 0.5 here in GraphSAR) into the block list. For blocks with more than half zero elements, we

[1]If the number of vertices cannot be divided by 8, we will add some (at most 7) independent vertices without any edge connecting to them. In this way, the number of vertices can be divided by 8.

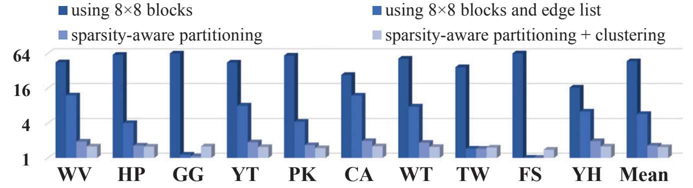

Figure 4: Memory space requirements when storing graphs in different ways (normalized to storing all edges into the edge list), graphs are from TABLE 1.

divide the block into 4 disjoint smaller blocks (e.g., a block of size 8×8 is divided into 4 disjoint blocks of size 4×4), and repeat steps above until these blocks are assigned to the block list or the edge list. In this way, we can ensure graphs are stored in the form of small adjacency matrices with more than half non-zero elements, or the edge list. We show the memory space requirement after adopting our sparsity-aware graph partitioning scheme in Figure 4. As we can see, only 1.63x memory space is required on average, compared with storing all edges into the edge list. In some cases (e.g., GG and FS), there is almost no extra memory space requirements, because nearly all blocks only contain 1 edge (Figure 3). We will further improve the result by using the clustering method.

3.3 Lightweight Graph Clustering

Empty subgraphs are skipped in GraphSAR (and in GraphR). Thus, clustering edges in a graph to reduce non-empty subgraphs is a promising way to improve the performance of GraphSAR and GraphR. Inspired by previous works [21, 22], that vertices appear together in the original edge list usually have locality, we propose our lightweight graph clustering method by remapping the index of each vertex. Figure 5 shows an example of this lightweight graph clustering method. When we load the original edge list, we assign the index to each new vertex from 0. Vertex 0 and 2 are the first two vertices in the edge list, then we map $0 \rightarrow 0$ and $2 \rightarrow 1$. After that, we process whole edge list in this way. In the example of Figure 5, we can reduce the number of non-empty subgraphs from 6 to 4.

The detailed results on real-world graphs are shown in Figure 6. It is shown that, by adopting the lightweight graph clustering method, the total number of subgraphs to be processed can be reduced by up to 70.38% (26.17% on average). Such index mapping can be maintained by a hash table, and processed at the same time of loading edges. Thus, the time complexity is only $O(\#edges)$. Although previous work [23] proposed graph clustering method, it was based on the Breadth-First Search algorithm. Such method has already performed graph algorithms when clustering, leading to heavy overheads. Moreover, as we can see in Figure 4, our clustering method

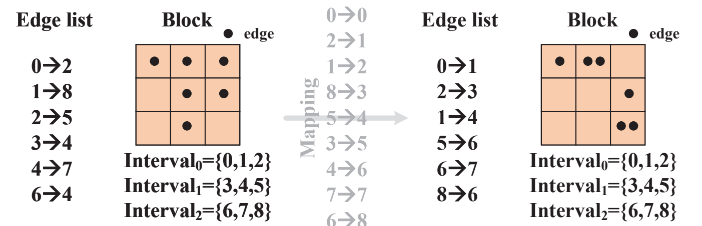

Figure 5: Clustering in GraphSAR, there are 9 vertices (dots) in the graph, each small orange square represents a block.

Figure 6: The number of processed subgraphs after clustering, normalized to number of w/o clustering.

can further reduce the memory requirements to 1.54x, compared with storing all edges using the edge list.

4 GraphSAR Architecture

4.1 Overall Architecture

By storing graphs using the block list and the edge list, GraphSAR can process subgraphs in the memory where these subgraphs are stored. Thus, GraphSAR is different from GraphR [1] and HyVE [19], by both computing and storing edge data in ReRAM crossbar. Figure 7 shows the top view of GraphSAR architecture. As we can see, we organize GraphSAR into banks with peripheral circuits, working as both memory and computation parts. Each bank is composed of multiple mats (ReRAM crossbars), and GraphSAR can perform matrix-vector multiplication based computation on these mats with peripheral circuits. The size of each mat can be 512×512 or 1024×1024 [24], while we can perform multiplication on matrices of smaller sizes (e.g., 4×4 or 8×8) by selecting wordlines and bitlines in a mat. The detailed functions of each component and peripheral circuits are described as follows:

- **ReRAM crossbar.** Edges are stored in ReRAM crossbars in both edge list and block list formats, shown on the right side of Figure 7. Edges and blocks of different sizes are stored on different banks for the alignment purpose. The detailed processing flow of two types of graph algorithms (matrix-vector multiplication and non-matrix-vector multiplication) is shown later.
- **Register file (Reg).** Both ReRAM crossbars and sALUs use data in the register file for computation. Vertex data are loaded to the register file during runtime. Note that **blocks in the block list sparsely spread over the graph, we can treat each block as an 'edge' connecting multiple vertices**. We have four types of 'edges', 8×8, 4×4, 2×2, and also original individual edges. We can treat these four types of 'edges' as four subgraphs separately (We do store blocks with different sizes separately for the alignment purpose, mentioned above). In this way, GraphSAR can adopt the similar scheduling scheme in GraphR with no extra scheduling/routing overheads.
- **Simple ALU (sALU).** The sALU is a simple customized algorithmic and logic unit, and it is at the output ports of ReRAM crossbar. We add 8 sALUs to each mat, and results from different bitlines can be selected and sent to these sALUs through multiplexers.
- **Scheduler.** The scheduler is responsible for vertex data loading and scheduling in GraphSAR. Because graphs are divided into blocks of different sizes, the scheduler is also responsible for selecting wordlines and bitlines.
- **Sample and Hold (S/H).** S/H samples and holds analog currents before converting into digital values.
- **Analog to Digital Converter (ADC).** ADC is used to convert analog data to digital data. We share ADC among bitlines, based on Previous works [1, 25].
- **Shift and Add unit (S/A).** GraphR has mentioned that the Shift and Add units are used to alleviate the pressure of driver, because the precision of ReRAM cells is low.

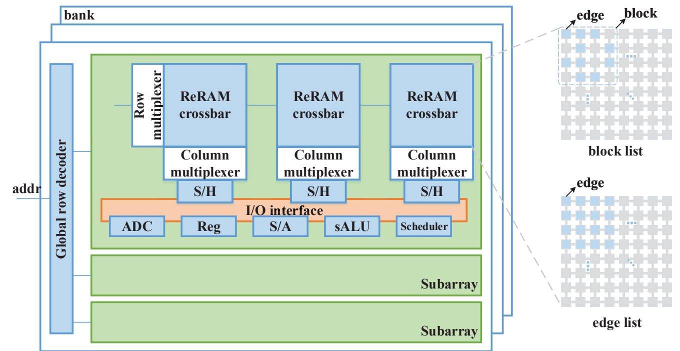

Figure 7: The detailed bank-level architecture of GraphSAR (left), and the data allocation of the edge list and the block list (right).

Compared with GraphR, GraphSAR can process graphs directly in the memory where graphs are stored, eliminating overheads of converting graphs from edge list into adjacency matrix. The processing-in-memory design can also provide *memory-capacity-proportional* [12, 25] computation units, rather than fixed number of processing units (graph engines) in GraphR.

4.2 GraphSAR Processing Flow

By using the sparsity-aware partitioning method, a graph is divided into blocks and edges. GraphSAR scan blocks in the block list and edges in the edge list to perform graph algorithms based on the hybrid-centric model. The order of blocks and edges in lists is the same as that in GraphR [1], where data are stored in the column-oriented order, leading to less usage of registers and write cost.

There are two main types of graph algorithms, matrix-vector multiplication based and non-matrix-vector multiplication based algorithms. GraphSAR processes them in different ways. We use PageRank and Breadth-First Search as examples of these two types of algorithms, to show the detailed processing flow in GraphSAR.

- **Matrix-vector multiplication based.** We take PageRank as the example. For blocks in the block list (blocks with multiple edges), GraphSAR first selects the 4×4 cells from the large crossbar based on peripheral circuits proposed in Fig. 7. Then, the value of each source vertex is sent to input ports of corresponding rows. The ReRAM crossbar performs the matrix-vector multiplication operation to calculate the rank value of each destination vertex. For edges in the edge list, GraphSAR just uses sALUs to calculate the rank value of the destination vertex.
- **Non-matrix-vector multiplication based.** We take Breadth-First Search as the example. For blocks in the block list,

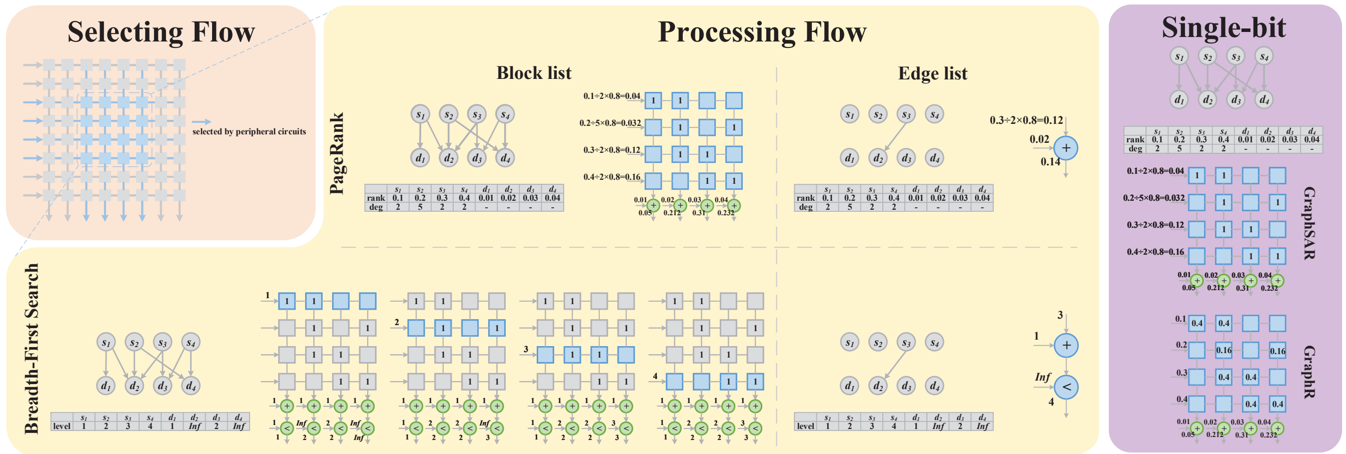

Figure 8: Detail processing flow of GraphSAR (left) and single-bit ReRAM cell implementation in GraphSAR (right). The left top shows an example of selecting a $4{\times}4$ from an $8{\times}8$ ReRAM crossbar. The rest part shows how GraphSAR processes subgraphs stored in the block list and the edge list, performing different algorithms. The size of processed matrix is set to 4×4 as an example, and the PageRank factor is set to 0.8.

GraphSAR activates each row sequentially. When a row is activated, the value of corresponding source vertex is transferred to neighbors by selecting corresponding columns, and the sALU on each output port makes the comparison operation between source and destination vertices. For edges in the edge list, GraphSAR also just uses sALU to calculate the rank value of the destination vertex.

4.3 Single-bit ReRAM Cell Implementation

We also show how to avoid requirements for high-precision ReRAM cells when running some algorithms. We take PageRank as an example on the right side of Figure 8 to show the difference between GraphSAR and GraphR. In PageRank algorithm, the rank value of each source vertex multiplied by a factor (divided by the out-degree and multiplied by the PageRank factor) before sending to the destination vertex. GraphR stores factors of vertices to corresponding rows in the ReRAM crossbar, and the rank value of each vertex is multiplied by the factor during processing on crossbars. Note that factors of a same source vertices to different destination vertices are same, thus we can directly store that value in the memory instead of calculating it every time.[2] In this way, GraphSAR only needs to store the connectivity of subgraphs using 1-bit cells. Such implementation is available for all unweighted graph algorithms, because all outgoing edges of the same source vertices propagate a same value, edges only represent the connectivity among vertices. Algorithms like Single Source Shortest Path are not applicable in this case because out-going edges of one vertex have different values.

5 Experimental Evaluation

5.1 Experimental Setup

5.1.1 Datasets and Algorithms. We choose 10 real-world graphs from different domains in our experiments, shown in TABLE 1. The number of vertices in these graphs ranges from thousands (k) to billions (b), and edges from millions (m) to billions (b). We implement 3 different graph algorithms. Breadth-First Search (BFS), PageRank (PR), and Connected Components (CC).

Table 1: Graph datasets used in evaluation

Datasets	#Vertices	#Edges	Type
wiki-Vote (WV) [26]	7.12 k	0.10 m	social network
cit-HepPh (HP) [26]	34.5 k	0.42 m	citation graph
web-Google (GG) [26]	0.88 m	5.11 m	web graph
com-Youtube (YT) [26]	1.13 m	2.99 m	community
soc-Pokec (PK) [26]	1.63 m	30.6 m	social network
roadNet-CA (CA) [26]	1.97 m	2.77 m	road network
wiki-Talk (WT) [26]	2.39 m	5.02 m	communication
twitter-2010 (TW) [27]	41.7 m	1.47 b	social network
com-Friendster (FS) [26]	65.6 m	1.81 b	community
yahoo-web (YH) [28]	1.41 b	6.64 b	web graph

5.1.2 Configurations. To evaluate the latency and energy consumption of GraphSAR, we choose NVSim [29] from many ReRAM simulators [29, 30]. We modify the model of ReRAM cell in NVSim. The parameter of ReRAM cell model is from the same source [20] in GraphR [1] (read/write energy consumption: 1.08pJ/7.4pJ, [3] read-/write latency: 29.31ns/50.88ns, HRS/LRS resistance: 25MΩ/50KΩ, read/write voltage: 0.7V/2V, current of LRS/HRS: 40μA/2μA). We model registers using Cacti [32], and the data of Analog/Digital converters are from [33]. For other logics, we evaluate the performance using code instrumentation to get the memory access trace,

[2] The calculation shown at the input port happens when GraphSAR loads values from the memory, in Figure 8.

[3] The write energy consumption used here (7.4 pJ) is much smaller than that used in GraphR (3.91nJ). The data used in GraphR are based on the whole ReRAM chip rather than ReRAM cell. We refer to [31] to get a much lower write energy consumption. Because GraphR needs to write ReRAM cells more frequently than GraphSAR, GraphR benefits more from lower write energy consumption.

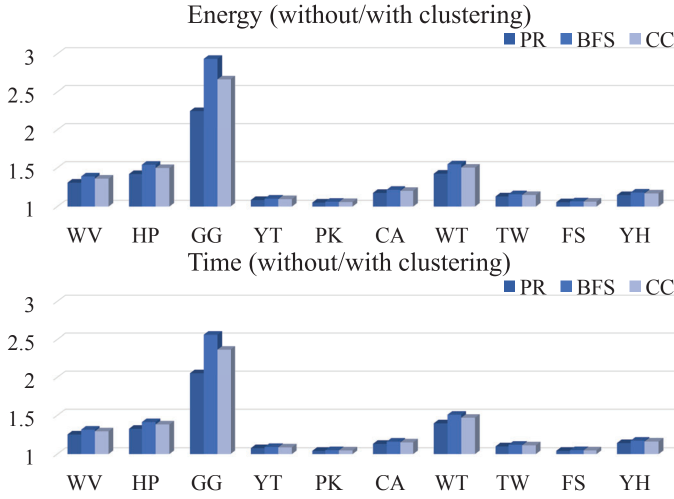

Figure 9: Benefit of lightweight graph clustering.

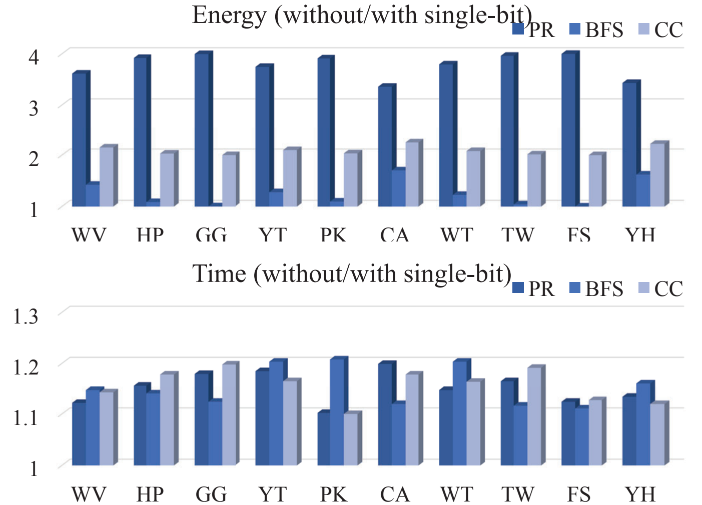

Figure 10: Benefit of using single-bit cell.

and the map the trace to corresponding device considering the model. For multi-level ReRAM cells, we modify NVSim according to the parallel sensing scheme proposed in [20], which enables the 'Write-and-verify' process of accessing multi-level ReRAM cells.

5.2 Benefits of GraphSAR Designs

5.2.1 Benefit of Clustering. We have proposed the lightweight graph clustering method to reduce the number of blocks to be processed. In this way, more blocks are skipped and less (but denser) blocks are processed.

Shown in Figure 9, using lightweight graph clustering method leads to averagely 1.37x energy reduction and 1.30x speedup (1.78x lower EDP). Graph GG benefits most from our clustering method, because over 99% non-empty blocks are with only one edge (shown in Figure 3). These blocks are merged and the total number non-empty blocks are reduced.

5.2.2 Benefit of Single-bit Implementation. By adopting such method, GraphSAR does not need to read/write multi-level ReRAM cells. Different from GraphR where multiple ReRAM crossbars with a Shift and Add units are used to calculate an edge stored in multi-bit cells, only one crossbar is required in GraphSAR.

As you can see in Figure 10, using our single-bit implementation leads to averagely 2.37x energy reduction and 1.15x speedup (2.73x lower EDP) compared with multi-bit implementation. Such implementation even results in higher energy reduction on some high-precision required algorithms (e.g., PR, 3.77x on average), because more ReRAM crossbars are required in these algorithms.

5.3 Comparison with GraphR

Based on the clustering and single-bit implementation mentioned in Section 5.2, the performance of GraphSAR is improved. Such methods can also be applied to GraphR to improve the performance of GraphR. Thus, in this section, we simulate GraphR with clustering and single-bit implementation, and compare it with GraphSAR.

As we can see in Figure 11. Compared with GraphR, GraphSAR achieves 4.43x energy reduction and 1.85x speedup (8.19x lower EDP). Such comparison can show performance improvements by adopting the sparsity-aware partitioning and processing-in-memory schemes.

5.4 Comparison with HyVE

Blocks can be treated as 'edges' in GraphSAR. HyVE is another graph processing accelerator using ReRAM, which processes graphs edge by edge. Compared with HyVE, GraphSAR can process multiple edges in a block. We compare the speedup and energy reduction between GraphSAR and HyVE, and list the ratio of edges in the block list of GraphSAR.

As we can see in Figure 12, GraphSAR achieves 1.29x speedup and 2.18x energy reduction compared with HyVE on average. Moreover, when there are more edges in the block list, the performance improvement becomes greater. GraphSAR outperforms HyVE by processing multiple edges simultaneously.

5.5 Design Space Exploration

As the emerging technology, ReRAM designs have been evolving year by year, and the parameters of ReRAMs can be quite different based on various experimental measurements. We have compared performance between GraphSAR and GraphR in previous sections using parameters mentioned in our evaluation setup. To further compare the performance between GraphSAR and GraphR on different ReRAM cell models, we modify NVsim by choosing different optimization targets, to get parameters of ReRAM in different situations. We calculate the average ratio of energy consumption and execution time when running PageRank on our datasets.

As we can see in TABLE 2, GraphSAR outperforms GraphR under different configurations. The reason is the GraphSAR eliminates overheads of writing ReRAM cells during processing in GraphR.

Table 2: Comparison under different configurations (improvement of GraphSAR compared with GraphR).

Opt.target	Time	Energy
Area optimized	1.31	2.18
Read latency optimized	19.40	22.52
Write latency optimized	9.53	10.07
Read energy optimized	6.77	174.68
Write energy optimized	6.34	149.75

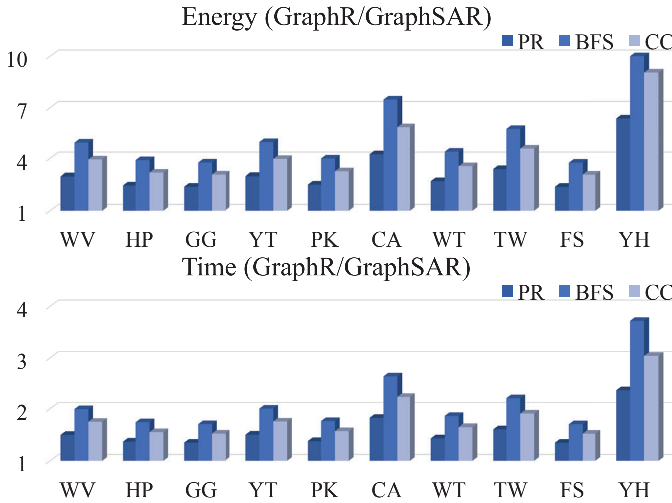

Figure 11: Comparison between GraphR and GraphSAR.

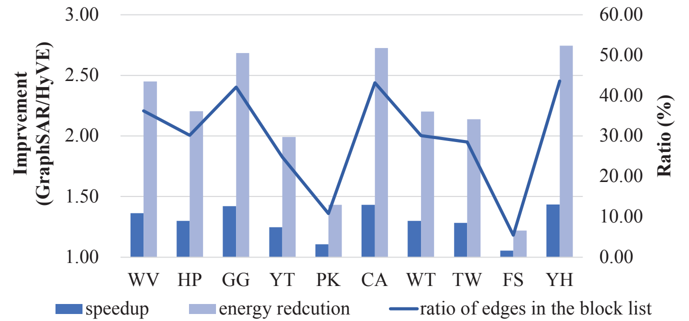

Figure 12: The relationship between the energy-efficiency and edges in blocks.

6 Conclusion

In this paper, we propose GraphSAR, a sparsity-aware processing-in-memory architecture for large-scale graph processing on ReRAMs. Frequent writing to ReRAMs leads to heavy overheads, especially when writing data to ReRAMs and only performing computation once over these data (e.g, GraphR [1]). Compared with GraphR, GraphSAR actually exploits the processing-in-memory feature of ReRAMs by directly processing edges in the memory, with only 1.54x memory space overheads. By removing overheads of writing data to ReRAMs, GraphSAR achieves 4.43x energy reduction and 1.85x speedup (8.19x lower energy-delay product, EDP) against GraphR.

7 Acknowledgement

This work was supported by Beijing National Research Center for Information Science and Technology (BNRist), Beijing Innovation Center for Future Chip, the project of Science and Technology on Reliability Physics and Application Technology of Electronic Component Laboratory (614280604011628O6001), National Key R&D Program of China 2017YFA0207600, and National Natural Science Foundation of China (61622403, 61621091).

References

[1] Linghao Song et al. Graphr: Accelerating graph processing using reram. In *HPCA*, pages 531–543. IEEE, 2018.
[2] Aapo Kyrola et al. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, pages 31–46. USENIX, 2012.
[3] Amitabha Roy et al. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488. ACM, 2013.
[4] Xiaowei Zhu et al. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *ATC*, pages 375–386. USENIX, 2015.
[5] Yuze Chi et al. Nxgraph: An efficient graph processing system on a single machine. In *ICDE*, pages 409–420. IEEE, 2016.
[6] Grzegorz Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.
[7] Yucheng Low et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.
[8] Tae Jun Ham et al. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *MICRO*, pages 1–13. IEEE, 2016.
[9] Guohao Dai et al. Fpgp: Graph processing framework on fpga a case study of breadth-first search. In *FPGA*, pages 105–110. ACM, 2016.
[10] Tayo Oguntebi et al. Graphops: A dataflow library for graph analytics acceleration. In *FPGA*, pages 111–117. ACM, 2016.
[11] Guohao Dai et al. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *FPGA*, pages 217–226. ACM, 2017.
[12] Junwhan Ahn et al. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*, pages 105–117. IEEE, 2015.
[13] Guohao Dai et al. Graphh: A processing-in-memory architecture for large-scale graph processing. *TCAD*, 2018.
[14] Mingxing Zhang et al. Graphp: Reducing communication of pim-based graph processing with efficient data partition. In *HPCA*, pages 544–557. IEEE, 2018.
[15] Fabien Alibart et al. High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm. *Nanotechnology*, 23(7):075201, 2012.
[16] Cong Xu et al. Overcoming the challenges of crossbar resistive memory architectures. In *HPCA*, pages 476–488. IEEE, 2015.
[17] Lixue Xia et al. Technological exploration of rram crossbar array for matrix-vector multiplication. *JCST*, 31(1):3–19, 2016.
[18] Lei Han et al. A novel reram-based processing-in-memory architecture for graph computing. In *NVMSA*, pages 1–6. IEEE, 2017.
[19] Tianhao Huang et al. Hyve: Hybrid vertex-edge memory hierarchy for energy-efficient graph processing. In *DATE*. EDA Consortium, 2018.
[20] Cong Xu et al. Understanding the trade-offs in multi-level cell reram memory design. In *DAC*, pages 1–6. IEEE, 2013.
[21] Xiaowei Zhu et al. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316. USENIX, 2016.
[22] Scott Beamer et al. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *IISWC*, pages 56–65. IEEE, 2015.
[23] Jianwei Cui and Qinru Qiu. Towards memristor based accelerator for sparse matrix vector multiplication. In *ISCAS*, pages 121–124. IEEE, 2016.
[24] Dimin Niu et al. Design of cross-point metal-oxide reram emphasizing reliability and cost. In *ICCAD*, pages 17–23. IEEE, 2013.
[25] Ping Chi et al. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ISCA*, pages 27–39. IEEE, 2016.
[26] Jure Leskovec et al. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.
[27] Haewoon Kwak et al. What is twitter, a social network or a news media? In *WWW*, pages 591–600. ACM, 2010.
[28] Yahoo WebScope. Yahoo! altavista web page hyper-link connectivity graph, circa 2002. http://webscope.sandbox.yahoo.com/, 2012.
[29] Xiangyu Dong et al. Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory. In *Emerging Memory Technologies*, pages 15–50. Springer, 2014.
[30] Lixue Xia et al. Mnsim: Simulation platform for memristor-based neuromorphic computing system. *TCAD*, 37(5):1009–1022, 2018.
[31] Shimeng Yu et al. Investigating the switching dynamics and multilevel capability of bipolar metal oxide resistive switching memory. *Applied Physics Letters*, 98(10):103514, 2011.
[32] Steven JE Wilton and Norman P Jouppi. Cacti: An enhanced cache access and cycle time model. *JSSC*, 31(5):677–688, 1996.
[33] Boris Murmann. Adc performance survey 1997-2017. http://web.stanford.edu/~murmann/adcsurvey.html, August 2017.