2023 Design, Automation & Test in Europe Conference (DATE 2023)

SparseMEM: Energy-efficient Design for In-memory Sparse-based Graph Processing

Mahdi Zahedi, Geert Custers, Taha Shahroodi, Georgi Gaydadjiev, Stephan Wong, Said Hamdioui
Department of Quantum and Computer Engineering, Delft University of Technology, Delft, The Netherlands
Email: {M.Z.Zahedi, G.A.J.Custers, T.Shahroodi, g.n.gaydadjiev, J.S.S.M.Wong, S.Hamdioui}@tudelft.nl

***Abstract*—Performing analysis on large graph datasets in an energy-efficient manner has posed a significant challenge; not only due to excessive data movements and poor locality, but also due to the non-optimal use of high sparsity of such datasets. The latter leads to a waste of resources as the computation is also performed on zero's operands which do not contribute to the final result. This paper designs a novel graph processing accelerator, SparseMEM, targeting sparse datasets by leveraging the computing-in-memory (CIM) concept; CIM is a promising solution to alleviate the overhead of data movement and the inherent poor locality of graph processing. The proposed solution stores the graph information in a compressed hierarchical format inside the memory and adjusts the workflow based on this new mapping. This vastly improves resource utilization, leading to higher energy and permanence efficiency. The experimental results demonstrate that SparseMEM outperforms a GPU-based platform and two state-of-the-art in-memory accelerators on speedup and energy efficiency by one and three orders of magnitude, respectively.**

***Index Terms*—in-memory, memristor, graph, sparsity**

I. Introduction

Graph processing is employed in a wide range of areas including but not limited to social media analysis [1], urban planning [2], and machine learning [3]. Graph processing is well-known for its three main characteristics [4], [5]: 1) poor locality or random access pattern to the memory, 2) simple and a small amount of computation over the accessed data, 3) high sparsity of the graphs which implies that the computations are frequently performed on zeros operands. Traditionally, the active portions of the graph are loaded sequentially into the memory hierarchy of the system while the rest of the graph is stored in the secondary memory. Due to the explosion of graphs' size, data movement across the memory hierarchy imposes a considerable overhead compared to the actual computation time/energy and limits the system's performance due to the maximum memory bandwidth. Moreover, some part of this latency and energy is wasted due to the sparsity of the graph. Clearly, there is a need for new architectures and methodologies to address the challenges mentioned above.

Several hardware accelerators were proposed to enhance the energy efficiency and performance of graph processing. Some of them have focused on optimizing memory access [6], others on improving the computational efficiency [7]. To further improve the system's efficiency beyond memory bandwidth limitation, near-memory computing was deployed in some previous works; e.g., TESSERACT [8] implements a vertex-centric programming model on top of Hybrid Memory Cube (HMC). To mitigate the communication overhead between different memory cubes, numerous solutions based on efficient graph partitioning [4], configurable interconnect [9], and batched-based communication [10] were provided. To further reduce the memory bandwidth limitation, GraphR [5], GRAM [11], and GraphSAR [12] proposed promising designs by exploiting computing-in-memory (CIM) based on emerging non-volatile memristors. Unlike the works mentioned earlier, GraphSAR [12] takes into account graph sparsity and provides a CIM sparsity-aware design on top of memristor devices in which sub-graphs with low density are divided into smaller ones. This approach can partially eliminate the sparsity and has a high pre-processing overhead. Hence, there is still a need for energy-efficient solutions that minimize the data movement overhead while taking the data sparsity into consideration.

In this paper, we propose SparseMEM, an energy-efficient design leveraging CIM; the design obtains maximum benefit from data sparsity. SparseMEM presents graph information in a hierarchical compressed format and comprises two key components: 1) Destination-Weight (DW) Crossbar, where the graph information is stored in a novel compressed representation; 2) Translation-Table (TT) Crossbar, which helps to navigate through the DW Crossbar. We implement the design using ReRAM memristor technology; memristor devices have great scalability, high density, near-zero standby power, and non-volatility [13], [14]. We compare SparseMEM with software implementation on a GPU platform as well as two CIM designs [5], [12]. The results show that we achieve $18\times$ speed up and $2000\times$ energy efficiency on average compared to the baselines. In short, our main contributions are:

- A novel data representation tailored for spars-based graph processing and targeting computing-in-memory designs. This enables the computations over a compressed graph representation (stored inside the memory) irrespective of the used memory technology;
- An optimized and scalable end-to-end ReRAM-based computing-in-memory accelerator that makes use of the proposed data representation for several widely used graph algorithms. The efficiency of the accelerator is studied over the different levels of graph sparsity;
- Case studies of different workloads to evaluate the design for different performance metrics. The design is compared with a software implementation on a high-end GPU platform and two in-memory state-of-the-art designs.

The paper is organized as follows. Section II provides background on memristor devices and graph processing. We discuss our SparseMEM proposal in Section III. Section IV evaluates the design, while Section V concludes the paper.

978-3-9819263-7-8/DATE23/© 2023 EDAA

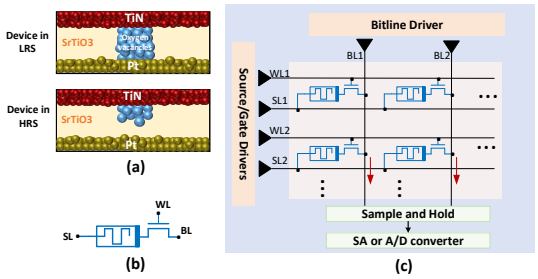

Fig. 1: (a) ReRAM memristor device behavior (b) 1T1R memristor cell (c) CIM tile encompassing crossbar and peripheries

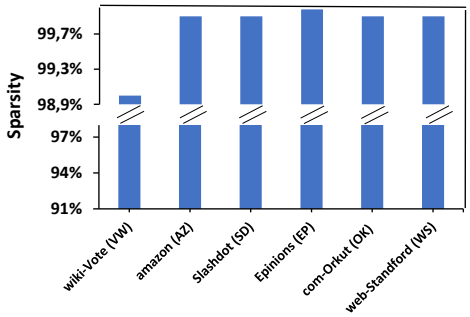

Fig. 2: Percentage of sparsity in adjacency matrix over some well-known graph datasets

II. Background

A. CIM based on Memristor devices

Despite charge-based memories, memristor devices hold data as resistance levels. The data can be presented as a binary value utilizing a low resistive state (LRS) and a high resistive state (HRS). Among different memristor technologies, Figure 1(a) illustrates Resistive Random-Access Memory (ReRAM) devices [15] consisting of a metal-insulator-metal stack; the bipolar device is set and reset by changing the polarity of the programming voltage (e.g., 2V) to form or dissolve the conducting filament. To read the device without disturbance, a small voltage (e.g., 0.2V) is applied, and the current (voltage) through (across) the device should be sensed while programming the device requires higher voltage/current and longer latency. Figure 1(b) shows a schematic representation of 1T1R memristor-based structure. This is a fundamental block for constructing a CIM tile encompassing memristors in crossbar structure and peripheries, as shown in Figure 1(c), where drivers are employed to drive Select-line (SL), Word-line (WL), and Bit-line(BL). The analog output of the crossbar is captured and converted to the digital domain using a sense amplifier (SA) or A/D converter (ADC).

B. Fundamentals of graph processing and motivation

A widely used method of graph representation is an algebraic representation. In this method, an *adjacency matrix* is constructed where each entry (i, j) represents an edge from source vertex i to destination vertex j. This representation allows for intuitive calculations using linear algebra operations. However, naive storage of this matrix in memory does not scale. The storage occupied by a 2D matrix grows quadratically, meaning that large graphs quickly occupy an impractical amount of memory. However, by monitoring the graph information stored in the memory, we observe that most of the matrix elements do not contribute to the result of the computation since they represent an edge that does not exist between destination and source nodes. Figure 2 presents the sparsity of some well-known datasets [16]; this illuminates the intensity of sparsity in adjacency representation. Operating on sparse data not only increases the memory requirements, but also brings the computational efficiency down. The solution in this paper aims to maximize computational efficiency and resource utilization while performing over sparse datasets.

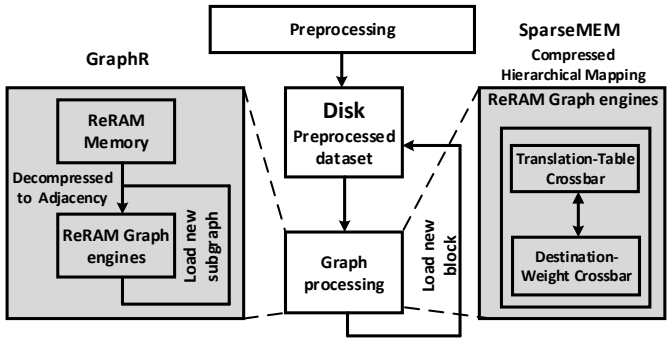

Fig. 3: SparseMEM and GraphR workflow comparison. Reducing the overhead of device programming by enabling computation over a compressed representation

III. SparseMEM Architecture

A. Overview of SparseMEM

Figure 3 shows the workflow of SpraseMEM compared to one of our baselines GraphR [5]. As stated before, real-world graph datasets are extremely large, even using compressed representations. However, the size of the storage unit, ReRAM memory, is practically limited due to the current technology restrictions. Therefore, we need to store the entire preprocessed graph dataset on disk. In the GraphR approach (as shown in Figure 3), the computation and storage are distinguished even within the ReRAM crossbars. While the first part (ReRAM Memory) holds the graph information loaded from the disk, the second part (ReRAM Graph engines) performs the computation over the uncompressed information. Nevertheless, the following challenges are faced: (a) Considerable data movement between ReRAM memory and Graph engines reduces the performance and energy efficiency of the system; (b) Conversion from the coordinate list (a compressed representation) to the adjacency representation (used for processing) imposes extra processing overhead on ReRAM engines in each iteration; and (c) Mapping the adjacency matrix to the graph engines leads to poor resource utilization due to high data sparsity. It is worth mentioning that increasing the number of memristor device programming reduces the endurance and energy efficiency.

SparseMEM presents the graph information in a new compressed *hierarchical* format inside the ReRAM crossbars.

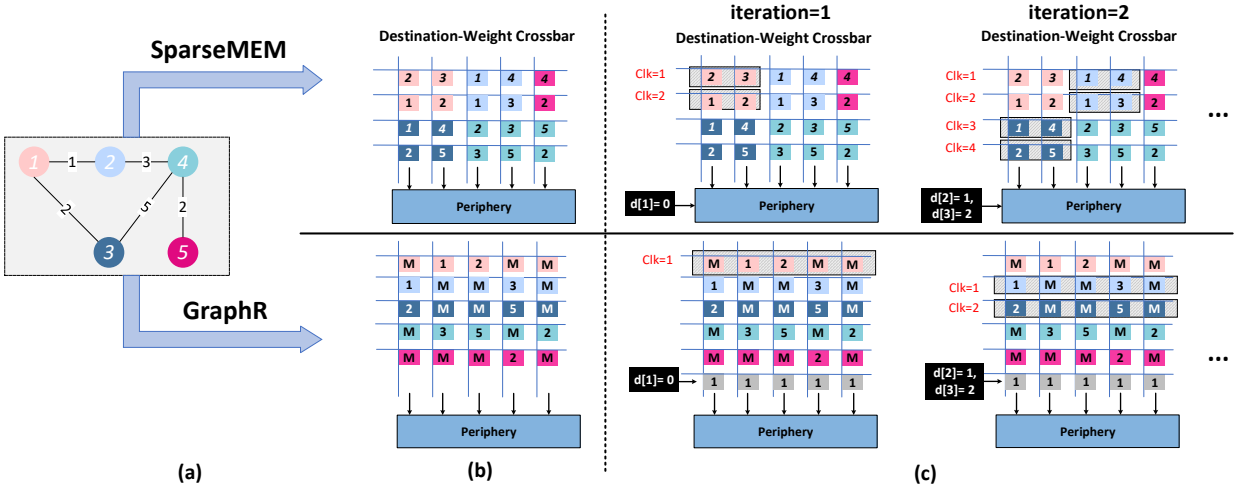

Fig. 4: (a) Graph example; (b) mapping of graph information into crossbar for computation phase considering SparseMEM and GraphR design; (c) the first two iterations of the SSSP algorithm starting with source vertex 1."M" means no connection

The design comprises two main components: 1) Destination-Weight (DW) Crossbar, where the graph information is stored in compressed representation 2) Translation-Table (TT) Crossbar, which decodes the information in the DW Crossbar and guides to extract information regarding the positioning of vertices which is needed for computation. This allows us to perform the computation exactly where the data is stored inside the ReRAM memory. Hence, there is no separation between ReRAM memory and the ReRAM processing unit. The SparseMEM major differentiators are: (a) it alleviates the number of data loading from disk due to the efficient use of ReRAM memory; (b) it uses computational resources efficiently by performing computation over compressed information; and (c) it eliminates the data conversion from compressed to adjacency representation.

B. Graph mapping and data representation

In this section, we explain our compressed hierarchical representation by providing a simple example based on a toy graph depicted in Figure 4(a). To clarify more on SparseMEM and identify the differences, we compare it with the GraphR design [5], where the adjacency matrix is directly mapped to the crossbar (similar to GraphSAR [12]).

In the case of GraphR (below part of Figure 4), where the adjacency matrix is directly mapped to the crossbar, the storage of edge weights encodes more information than just the weight. Since the entries are expanded, the location of the edge in the crossbar also encodes the source and destination vertex of this edge. Figure 4(b) shows the mapping used in GraphR. As an example, the *first* row of the crossbar shows which vertices of the graph are connected to vertex 1, the *second* row in the crossbar gives the vertices connected to vertex 2, etc. For example, vertex 1 is connected to vertices 2 and 3; entities in the matrix give the weight of connections (1 and 2) and "M" denotes no connection. However, this information is lost when the edges are stored in a compressed format. Thus, a proper mapping to preserve this information is required.

In the case of SparseMEM design (top part of Figure 4), optimal use of storage is made. Each graph's vertex gets a *sub-array* in the **Destination-Weight (DW) Crossbar**; each of these sub-arrays consists of two rows: one for the index of *destination* vertices and one for the *weights* of the edges connecting the source to destination vertices; the number of columns in a sub-array depends on the connectivity of a vertex to other vertices. In Figure 4(b), each color in the DW Crossbar represents a sub-array for a vertex. For example, the pink color presents the sub-array associated with vertex 1. As vertices 2 and 3 are connected to vertex 1, we store the index of these vertices in the *first* row (being 2 and 3) and their weights (being 1 and 2 respectively) in the *second* row. Note that, in SparseMEM, only the non-sparse data is stored which is required for computation. E.g., for vertex 1 only four values are stored in DW Crossbar. However, in the GraphR design, the entire row 1, which represents the collection of edges with source vertex 1, has to be stored. All these devices contribute to the execution, even though only two hold the data of interest and the rest hold a predefined value representing no connection.

The key question is now how to preserve the information regarding the location of edges belonging to a vertex in the DW Crossbar. In order to encode this information, a separate **Translation Table (TT) Crossbar** is used. This crossbar is employed to encode the location of the edges of a particular source vertex (being stored in the DW Crossbar) and navigates through it. In the TT Crossbar, we store the information for each vertex as 'start address' and 'end address'; these refer to the first and last location, respectively, occupied by a vertex in the DW Crossbar. Figure 5 illustrates an example of 4×4 TT Crossbar; it assumes the addressing is performed in an increment manner from left to right (i.e., fast column addressing). The first two addresses are reserved for vertex 1, the second two addresses for vertex 2, etc. For example, the start and end addresses of vertex 3 stored in the TT Crossbar are 6 and 7, respectively. To translate the address

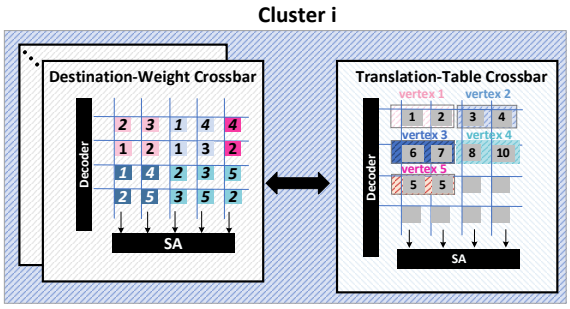

Fig. 5: Mapping of information regarding positioning of vertices in the DW Crossbar into the TT Crossbar

to the DW Crossbar addresses, the following formula is used: $A_{DW} = A_{TT} + \lfloor A_{TT}/C \rfloor \times C$ where C denotes the number of columns in the DW Crossbar and $\lfloor\rfloor$ denotes the floor of division. In the example of Figure 5, C=5; hence, for vertex 3, A_{DW} is 11 (start) and 12 (end), assuming also fast column addressing of DW Crossbar. Note that, the translation of A_{TT} to A_{DW} skips the even rows as they always store the weights corresponding to A_{DW} addresses (vertices). This hierarchical storage format eliminates sparsity, while still preserving edge source and destination. TT Crossbar is operating in parallel with DW Crossbar and can provide information to multiple DW Crossbars regarding the address of active vertices for the next iteration. Several DW Crossbars together with their TT Crossbar form a cluster. A design may contain several clusters.

C. Execution flow

In this subsection, we explain the execution flow of SparseMEM by providing an example based on Single-Source Shortest-Path (SSSP) algorithm (see Algorithm 1). In graph algorithms, the execution can be divided into two steps: a) compute and b) update. Considering the SSSP algorithm, we take a single start vertex and compute the distances "d" to every other vertex in the graph. The vertices whose distance values are updated in the current iteration are activated for the next iteration. The algorithm continues until there are no active vertices. When we access the edge that connects an active source vertex to a destination vertex, the edge weight has to be added to the current distance value of the source vertex (compute step). At the end, we need to update the destination registers (update step) where we store the distance value of vertices to the source vertex. This is usually a simple mathematical operator and varies across algorithms. In the case of the SSSP algorithm, this operator is a *Min* function that gives the minimum of the current computed distance value as well as the old value for a vertex. Next, we illustrate the algorithm for SparseMEM and GraphR implementation.

GraphR: Figure 4(c) illustrates the first two iterations (line 3 in Algorithm 1) of the SSSP algorithm where we want to find out the distance of vertex 1 to other vertices. To clarify more on the implementation, we consider the second iteration of the algorithm in Figure 4(c) as an example where we want to find the distance from vertex 1 to the rest of the vertices through vertex 2 and 3 (they got activated after the

Algorithm 1 SSSP algorithm

```
1:  ActiveVertices[start] ← True
2:  d[start] ← 0
3:  while ActiveVertices ≠ ∅ do
4:      for v ∈ ActiveVertices do
5:          for each neighbour u of v do
6:              New_d[u] ← d[v] + weight_(v)(u) // compute
7:              d[u] ← min(New_d[u], d[u]) // update
8:              if d[u] changed then
9:                  ActiveVertices[u] ← True
10:             end if
11:         end for
12:     end for
13: end while
```

first iteration). When we get access to the edges belonging to vertex 2 (i.e., $weight_2[4] = 3$), the current distance from vertex 1 to vertex 2 ($d[2]$), which was computed in the first iteration, has to be added up ($New\_d[4] = d[2] + weight_2(4)$). In the GraphR implementation, this addition is performed in an analog manner where the current distance of the active vertex $d[2]$ is given to the crossbar as input as shown in the bottom part of Figure 4(c). Note that, the last row is programmed to value 1. This is because $d[2]$ has to be first multiplied by 1 and then added to the second row storing the edge weights belonging to vertex 2 (e.g., $weight_2[4]$). Due to the limited resolution of the input drivers, input data has to be sliced and applied to the crossbar in several steps. As a consequence, the implementation of such addition requires costly peripheral components like power-hungry ADC and Shift-and-Add units, as shown in Figure 6(a). After the compute step, the update step takes place in the periphery of the DW Crossbar using a digital circuit (Min in Figure 6), and stores the result in the destination register dedicated to this vertex.

SparseMEM: To demonstrate how SparseMEM performs computing, we consider the same case as we did for GraphR; i.e., the second iteration of Algorithm 1 where we aim at finding the distance from vertex 1 to vertex 4 through vertex 2. The compute step (i.e., $New\_d[4] = d[2] + weight_2(4)$) is calculated by first reading $weight_2(4)$ from DW Crossbar (see iteration 2 in Figure 4) and then adding it to $d[2]$, which is provided as input to the periphery of the crossbar. The addition is done with a digital adder as shown in Figure 6(b). This approach avoids activating the crossbar several times and using Shift-and-Add units. This also helps to replace expensive ADCs with simple Sense Amplifiers (SAs). However, SparseMEM needs a bus or an interconnect component. Due to the compressed representation in SparseMEM, the crossbar no longer encodes vertex location. Therefore, a bus is placed in order to navigate data to the target destination register. As already mentioned, before computing the new distance value (e.g., $New\_d[4]$), the index of the vertex where this value belongs (e.g., 4) is read from the DW crossbar. This information is used to configure the bus and navigate the new distance value to its destination register.

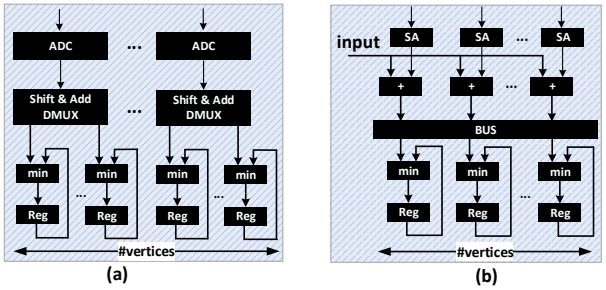

Fig. 6: Periphery design for (a) GraphR and (b) SparseMEM

D. Sub-graphs streaming and processing

To process graphs much larger than the available memory size provided by the crossbars, it is necessary to stream the graph data into the crossbar from another source, such as secondary memory storage. This is considered for GraphR, GraphSAR, and SparseMEM. To stream a graph, we split it up into "sub-graphs". To be more specific, the adjacency matrix is partitioned into sub-matrices which either represent the connectivity within a sub-graph or between two sub-graphs. In the case of GraphSAR [12] and GraphR [5] designs, sub-matrices in which all elements are equal to zero are eliminated from the process to improve efficiency in the presence of high data sparsity. In SparseMEM, while the sub-matrices are streamed from the secondary memory storage to the DW Crossbars, we program the TT Crossbar as well. This information is known at compile time and does not require extra processing during the execution. Such a format of graph streaming allows for processing graphs much larger than the available memory size provided by the crossbars.

IV. Evaluation and discussion

A. Experimental setup

Our simulation results are obtained by creating a platform for SparseMEM written in C++ [17], which takes graph datasets and performs the same steps as the hardware. These steps include the various operations performed on the crossbar, such as reading and writing to the crossbar. The parameters for ReRAM technology are taken from [18]. We assume each memristor cell can hold one bit (two resistance levels) for all the simulations. The parameters for ADCs are taken from [19] given in 32 nm technology, and the resolution of the input drivers for the crossbars are 1-bit [20]. We summarize the parameters regarding the crossbar technology in Table I. Digital peripheries are synthesized in Cadence Genus targeting standard cell 15 nm Nangate library. Finally, we use 16-bit integer data size in all experiments.

We evaluate SparseMEM in terms of speedup and energy while comparing the solution with GraphR [5], GraphSAR [12], and Nvidia Geforce RTX2080 GPU platform. We use Nvidia-smi to obtain the power consumption of the GPU platform. Our experiments concern three algorithms (application): 1) SSSP, 2) Breadth First Search (BFS), and 3) PageRank on real-world graph data sets, which are retrieved from the SNAP graph repository [16]. Datasets (workload) used for the experiments are summarized in Table II.

TABLE I: Memristor tile specification

Crossbar - ReRAM (128x128 @1bit)		
	Energy (Single Cell)	Latency
Read	40fJ	10ns
Write	20pJ	100ns
ADC (8-bit)		
Energy	2pJ per sample	
Latency	1ns per sample	
Shared with	32 columns	
SA		
Energy	0.01pJ per sample	
Latency	1ns per sample	
Shared with	4 columns	

TABLE II: List of graph datasets

Dataset	Average Degree	#Vertices	#Edges	Domain
wiki-Vote (WV)	29	7k	104k	Social
amazon0302 (AZ)	9	262k	1.23M	Co-purchasing
soc-Slashdot0902 (SD)	23	82k	948k	Social
soc-Epinions1 (EP)	13	75k	508k	Social
com-Orkut (OK)	39	3M	117M	Communities
web-Stanford (WS)	16	281k	2.3M	Web

B. Experimental Results

Figure 7 depicts speedup and energy improvements of SparseMEM w.r.t the baselines. It shows the following:

1) Speedup is strongly application and workload-dependent. E.g., SparsMEM archives minimum speedup for PageRank, and minimum speedup for 'WV' workload.
2) SparseMEM systematically outperforms in terms of energy efficiency irrespective of the workload and application.
3) On the average, SparseMEM achieves $18\times$ speedup and $2000\times$ energy improvement compared to CIM baselines.

Speedup: Sparsity is the main factor determining the speedup in SparseMEM compared to the baselines. Figure 8 depicts the utilization of memristor devices in SparseMEM and GraphR implementations; this represents the total number of (re)programming memristor devices during the execution of two algorithms. Less reprogramming devices in SparseMEM leads to significant improvement in speedup. According to the results, the minimum speedup improvement is for the Wiki-Vote (WV) workload with the lowest sparsity (see Figure 2); as sparsity reduces, the overhead of hierarchical mapping used in SparseMEM increases. Note that even if graphs have similar sparsity (e.g., 'AZ' and 'SD' workloads in Figure 2), the speedup can be different (see Figure 7); this is due to the distribution of data over sub-graphs (graph connectivity). If the data is scattered over many sub-graphs, there would be fewer sub-graphs whose all elements are zero. Consequently, there are fewer sub-graphs to be eliminated from the process, which increases the overhead in the baselines. Among applications, SparseMEM achieves similar (or less) performance to the baselines for the PageRank algorithm. Since the baselines directly map the adjacency matrix to the crossbar, analog matrix multiplication can be supported inside the crossbar. Therefore, they can achieve more parallelization for PageRank, where matrix multiplication is an essential kernel.

Energy: Sparsity and the periphery design are also major factors influencing energy consumption. As stated in sub-section III-C, the input drivers and ADCs are major energy

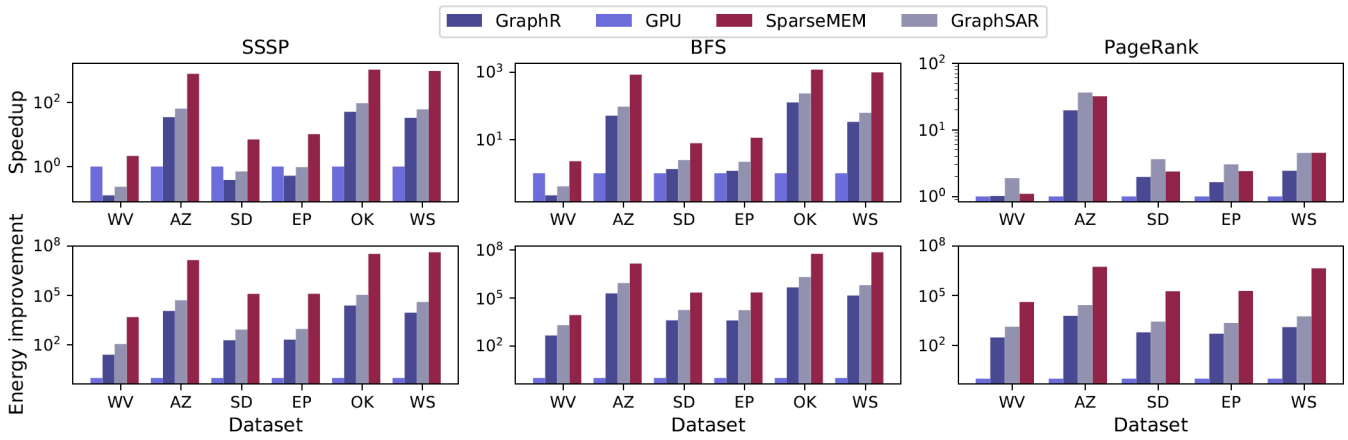

Fig. 7: Speedup and energy improvement of SparseMEM compared to the baselines for three algorithms (normalized to GPU)

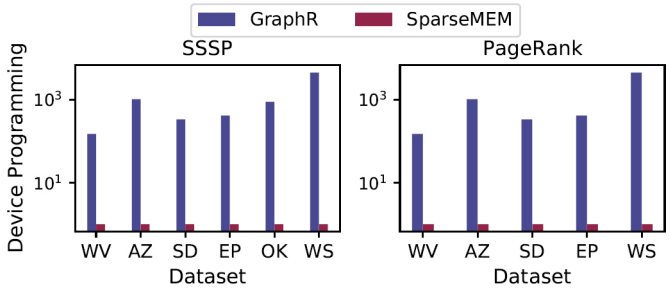

Fig. 8: Relative number of memristor programmings required by SparseMEM and GraphR (normalized to SparseMEM)

consumers. In GraphR implementation, feeding a data operand (e.g., 16 bits size) to the input driver requires several (e.g., 16) iterations due to the limited driver resolution [21]. This increases both energy and latency. In addition, using ADC in the periphery consumes more energy. Hence, higher energy overhead is imposed on the system. SparseMEM, however, is a read-driven design where the computation (e.g., addition) is performed in the periphery. This reduces the energy required for computation in the crossbar as well as its periphery.

V. Conclusion

This paper proposes SparseMEM, an in-memory graph processing accelerator tailored for sparse workloads. The key idea is the compressed and hierarchical mapping of the graph information into memristor-based crossbars to efficiently perform the computation inside the memory. The design requires less reprogramming of memristor devices (i.e., read-driven), while performing data-centric computing (CIM). The result implies the importance of hardware/mapping co-design for energy-efficient CIM design.

References

[1] I. Pitas, *Graph-based social media analysis*. CRC Press, 2016, vol. 39.

[2] H. Peng *et al.*, "Spatial temporal incidence dynamic graph neural networks for traffic flow forecasting," *Information Sciences*, vol. 521, pp. 277–290, 2020.

[3] X. Dong *et al.*, "Graph signal processing for machine learning: A review and new perspectives," *IEEE Signal processing magazine*, vol. 37, no. 6, pp. 117–127, 2020.

[4] M. Zhang *et al.*, "GraphP: Reducing communication for PIM-based graph processing with efficient data partition," in *HPCA*. IEEE, 2018, pp. 544–557.

[5] L. Song *et al.*, "GraphR: Accelerating Graph Processing Using ReRAM," in *HPCA*, 2018, pp. 531–543.

[6] J. Lin *et al.*, "Overcoming the Memory Hierarchy Inefficiencies in Graph Processing Applications," in *IEEE/ACM ICCAD*. IEEE, 2021, pp. 1–9.

[7] S. Rahman *et al.*, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020.

[8] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.

[9] G. Dai *et al.*, "Graphh: A processing-in-memory architecture for large-scale graph processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 640–653, 2018.

[10] Y. Zhuo *et al.*, "Graphq: Scalable PIM-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 712–725.

[11] M. Zhou *et al.*, "Gram: graph processing in a ReRAM-based computational memory," in *IEEE Asia and South Pacific Design Automation Conference*, 2019.

[12] G. Dai *et al.*, "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*, 2019, pp. 120–126.

[13] M. Zahedi *et al.*, "MNEMOSENE: Tile Architecture and Simulator for Memristor-based Computation-in-memory," *ACM JETC*, vol. 18, no. 3, pp. 1–24, 2022.

[14] M. Zahedi, "System Design for Computation-in-Memory: From Primitive to Complex Functions," in *IFIP/IEEE VLSI-SoC*. IEEE, 2022, pp. 1–6.

[15] O. Golonzka *et al.*, "Non-volatile RRAM embedded into 22FFL FinFET technology," in *2019 Symposium on VLSI Technology*. IEEE, 2019, pp. T230–T231.

[16] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[17] "SparseMEM simulation platform." [Online]. Available: https://github.com/Geertiebear/honours

[18] K. Fleck *et al.*, "Energy dissipation during pulsed switching of strontium-titanate based resistive switching memory devices," in *ESSDERC*. IEEE, 2016, pp. 160–163.

[19] L. Kull *et al.*, "A 3.1 mW 8b 1.2 GS/s single-channel asynchronous SAR ADC with alternate comparators for enhanced speed in 32 nm digital SOI CMOS," *IEEE JSSC*, vol. 48, no. 12, pp. 3049–3058, 2013.

[20] M. Saberi *et al.*, "Analysis of power consumption and linearity in capacitive digital-to-analog converters used in successive approximation ADCs," *IEEE TCAS I: Regular Papers*, vol. 58, no. 8, pp. 1736–1748, 2011.

[21] M. Zahedi *et al.*, "Efficient organization of digital periphery to support integer datatype for memristor-based CIM," in *2020 ISVLSI*. IEEE, 2020, pp. 216–221.