A Novel ReRAM-based Processing-in-Memory Architecture for Graph Computing

Lei Han† Zhaoyan Shen† Zili Shao† H. Howie Huang‡ Tao Li*
†Department of Computing, Hong Kong Polytechnic University
‡Department of Electrical and Computer Engineering, George Washington University
*Department of Electrical and Computer Engineering, University of Florida

***Abstract*—Graph algorithms such as breadth-first search (BFS) have been gaining ever-increasing importance in the era of Big Data. However, the memory bandwidth remains the key performance bottleneck for graph processing. To address this problem, we utilize processing-in-memory (PIM), combined with non-volatile metal-oxide resistive random access memory (ReRAM), to improve the performance of both computation and I/O. The idea is to integrate the computation logic into the memory in which the data accesses are located. We propose and implement a new ReRAM-based processing-in-memory architecture called RPBFS, in which graphs can be processed and persistently stored. We also design an efficient graph traversal scheme. Benefited from low data movement overhead and bank-level parallel computation, RPBFS shows a significant performance improvement compared with both the CPU-based and GPU-based BFS implementations. On a suite of real world graphs, our architecture yields up to 33.8× speedup.**

I. Introduction

The use of graph-based computation is ubiquitous in analyzing and understanding social networks, complex engineering systems and metabolic networks. Common graph-theoretic algorithms such as graph traversal have been gaining ever-increasing importance in these application areas. However, data movement between the processor and memory limits graph computation performance due to irregular memory accesses. Recently processing-in-memory techniques show great potential to address the memory wall challenge by integrating the computation logic within or near memory, thereby minimizing the speed mismatch for data-intensive applications.

Metal-oxide resistive random access memory (ReRAM) crossbar array, which can perform both computation and memory functions, is attractive, especially for matrix-vector multiplication. Considering the fact that the size of cells keeps shrinking, enhancing the density of ReRAM memory has been studied [1]. Furthermore, ReRAM has been considered as main memory thanks to the low read latency, higher endurance, and energy efficiency [2].

We study the problem of graph traversal in this work. In particular, breadth-first search (BFS) refers to a method of exploring all the vertices in a graph. Although very simple, it is challenging for BFS to achieve good performance in traditional computation system due to the fact that current architectures heavily penalize memory-intensive random accesses. The memory bandwidth is the bottleneck that hinders the graph traversal performance. In [3], the authors demonstrate that increasing computation cores is inefficient because higher performance would require bigger memory bandwidth. GPU can offer massive parallelism for high-performance graph computing but it needs to utilize memory hierarchy well. Accessing a shared memory is several orders of magnitude faster than that on a global memory. Beyond that, BFS implementation needs additional data structures. In recent GPU works [4] [5] [6], indispensable vertex arrays are maintained in global memory since the inspection data of the preceding level can be presented anywhere, which aggravates the amount of memory access.

Performance limitations of traditional graph processing platforms have been studied recently. [7] [8] have shown that the graph processing on optimized conventional systems results in a low IPC value. The main bottleneck, memory access, which suffers from poor temporal locality, has been identified. To maximize the available memory bandwidth, [3] integrates PIM technology into 3D-stacked memory. [9] proposes an accelerator architecture to reduce the irregular access patterns and asymmetric convergence. Although they are architectural accelerators for graph analysis, they can not make graphs persistently stored and ignore the optimization on graph layout.

In this work, we propose a novel ReRAM-based main memory PIM architecture (RPBFS) that is optimized for graph travel problems. In this architecture, a portion of memory banks called graph banks are used to store compressed sparse graph data, while a master bank is selected to record the metadata of graph banks and manage the graph traversal procedure. Due to the non-volatility of ReRAM, a graph is persistently stored and scattered over multiple graph banks. To accelerate graph traversal and keep it accurate, we propose an efficient scheme that can alleviate the impact of data movement on traversal performance. In our design, the accesses to adjacent vertex are constrained in the corresponding graph bank. The data movement in the internal bus among memory banks is only related to the vertex bitmap, which is smaller compared with other solutions. We have conducted a series of experiments and the results show significant improvement across a wide variety of graphs compared with the state-of-the-art CPU-based and GPU-based parallel solutions.

To the best of our knowledge, this is the first work to explore graph traversal on ReRAM crossbars, which we believe can benefit other graph algorithms, such as single-source shortest

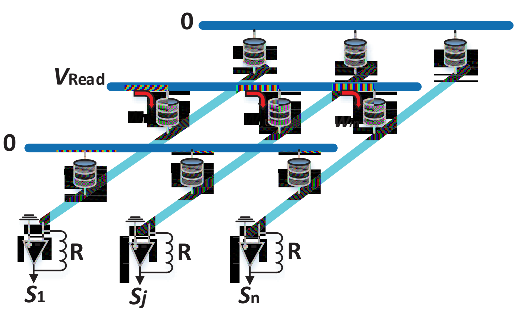

Fig. 1: Overview of ReRAM crossbar.

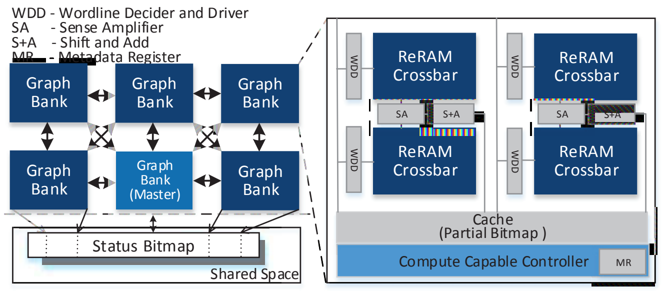

Fig. 2: RPBFS architecture.

path which can be solved by BFS. We conclude our contributions in this paper as follows:

- We design a new ReRAM-based main memory architecture with a set of peripheral circuits. ReRAM-based memory banks are separated to graph banks and master bank for accelerating graph traversal.
- We develop a novel mapping layout to store graph on ReRAM crossbars and to maximize computation capability.
- We design algorithms for graph bank and master bank, respectively. Data movement is minimized and bank-level parallelism is explored to effectively accelerate graph traversal.
- We evaluate our proposed scheme with a variety of real world graphs and the results show the performance improvement of 33.8× compared with the state-of-the-art CPU-based solution and 16.0× with the GPU-based solution.

II. Preliminaries

A. Breadth-First Search

Breadth-First Search starts the traversal from a given source vertex and systematically explores a graph to discover every vertex. According to frontiers queue initialized with the source vertex, BFS explores their adjacent vertices and marks them as visited with shortest depth.

Top-down BFS aiming to identify unvisited adjacent vertices of frontiers is a traditional traversal algorithm. The direction from top-down can be switched to bottom-up in a later state, which performs traversal more efficiently when the current frontiers are large [10] [4].

B. ReRAM Basics

A metal-oxide ReRAM cell consists of a top metal electrode, a metal-oxide resistive switch, and a bottom electrode. An ReRAM array can be interconnected as a dense crossbar architecture without transistors, which suits better for main memory due to the small area size of ReRAM cell. Figure 1 shows an area-efficient ReRAM crossbar array. If the input voltages V_1, V_2, ..., V_n are applied on the wordlines in the arrays, and the conductances $W_{i,j}$ of cells are programmed, the current S_j at the end of jth bitline represents the sum result of dot product operations, $\sum V_i \cdot W_{i,j}$. In our design, if the voltage of the selected wordline is set to 1, while others are set to 0, the results in bitlines are the exact conductances of cells which can identify adjacency vertices.

C. Processing-In-Memory

Processing-in-memory closely couples storage and computation capability to alleviate the bandwidth bottleneck between logic and memory. Moving computation logic to the place where data resides can reduce the energy cost as well. Driven by the 3D-stacking technology in recent years, PIM is resurgent by putting logic layer into 3D stacked memories [3]. Recently, with the development of emerging memory, the inherent computation capability of non-volatile memory can be directly integrated into the existing circuit design to implement in-memory computing without 3D-stacking technology [11].

III. ReRAM-based PIM Architecture for BFS

A novel ReRAM-based processing-in-memory architecture for breadth-first search (RPBFS), which can efficiently accelerate graph traversal by leveraging data layout in crossbar arrays and the PIM architecture, is proposed. Figure 2 depicts an overview of our design. Our architecture partitions ReRAM-based memory bank into two types: master bank and graph bank.

Master bank can be arbitrarily selected from memory banks to schedule expansion task for breadth-first traversal. Due to the limited size of ReRAM crossbars, a graph could hardly be stored in a single memory bank so that multiple graph banks are involved. The master bank stores corresponding metadata of graph banks, including vertex range information, and the starting row number of ReRAM crossbars.

A graph with several millions vertices and edges is mapped to multiple graph banks. Normally a graph can be represented with the Compressed Sparse Row (CSR) format to accelerate arithmetic operations. In order to keep the advantage of this format, the graph bank stores the adjacency list of CSR in contiguous cells. In this way, the adjacent vertices of one vertex are mapped to multiple cells one by one in ReRAM crossbar arrays. The adjacent vertices of one vertex may involve multiple rows if it is a hub vertex whose out-degree is large [4]. A pointer is needed to indicate row index and column index of adjacent vertices in ReRAM crossbar, which is similar to the indices pointer in the CSR format. We use location pointer [*row index, column index*] to replace the original indices pointer in order to identify the adjacency list in crossbar array. Both of adjacent list and location pointers are stored in one crossbar to keep correspondence. This way, the corresponding starting row number of adjacent list and location pointers of a crossbar array in each graph bank are recorded by the master bank.

This section describes the details of RPBFS microarchitecture and discusses how to map a graph on ReRAM crossbar.

A. *Microarchitecture*

A ReRAM-based main memory chip is composed of a number of memory banks, as shown in Figure 2. All the banks are interconnected in a Mesh network way for on-chip communication. They share one EDRAM (Enhanced dynamic random access memory) that stores the status bitmap of all the vertices in an expansion level. In each memory bank, an integrated controller with computing capability is used to decode instructions and provide control signals to all the peripheral circuits. Intermediate data is stored on the cache in memory banks. In our RPBFS architecture, the difference between graph bank and master bank is that the controller in graph bank maintains its own partition of frontier bitmap and and status bitmap, while master records the sum of the newly-visited vertices for scheduling. To efficiently traverse a graph, a number of digital components should be orchestrated with ReRAM crossbars. WDD is the wordline decoder and writing driver, which is used to access graph data. Sense amplifier (SA) implements the similar function of ADC, which can provide high precision control. Two ReRAM crossbars share one SA in order to reduce area overhead. Due to the limited precision of ReRAM cell, shift-and-adds (S+A) is provided to support higher precision for large-scale graph. The add results are then sent to the controller to determine whether these vertices have been traversed or not.

B. *Storing Graph in ReRAM Crossbar Array*

In each graph bank, a partition of adjacency list on a graph with the corresponding location pointers are stored in ReRAM crossbars. In RPBFS architecture, the adjacency list of one vertex is mapped to ReRAM cells one by one. The location pointers including row index and column index are used to indicate the corresponding adjacent vertices. We map the corresponding location pointers below to adjacency list and keep correspondence between them in a crossbar. Figure 3 shows an example of graph G and its CSR representation. Suppose that there are 6×6 ReRAM cells in one crossbar, and each cell can store one vertex, the first memory bank in Figure 4 could therefore demonstrate the mapping data from $Vertex$ 0 to $Vertex$ 5. We map adjacency list of this graph to crossbar arrays in a matrix way (indicated within red line). Location pointers are stored every two cells for row and column index, respectively. $Vertex$ 5 has five adjacent vertices (indicated by red color), but the third row in ReRAM crossbar does not have enough cells to store all of them, so its adjacency $Vertex$ 6 and $Vertex$ 9 are stored in the fourth row. The last two rows store the corresponding location pointers in this crossbar. The location pointer of $Vertex$ 5 can be attained by calculating the offset from the starting location pointer: $[(Vertex\ number \times 2)/row\_size + starting\_row, (Vertex\ number \times 2)\%column\_size]$, where row_size and $column\_size$ refer to the dimensions of cells in the ReRAM crossbar, $starting\_row$ refers to the starting row number of the location pointers. In order to completely obtain the adjacent vertices of $Vertex$ 5, it also needs to get the location pointer of the prior vertex. The location pointer of $Vertex$ 4 is $[2,\ 2]$, so the cells after location $[2,\ 2]$ to $[3,\ 1]$ are adjacent vertices of $Vertex$ 5.

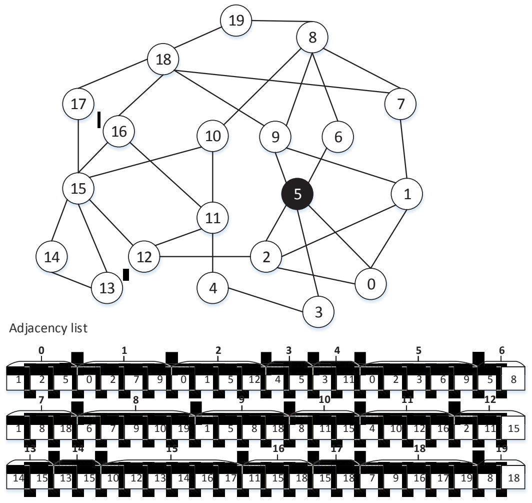

Fig. 3: An example graph G with adjacency list.

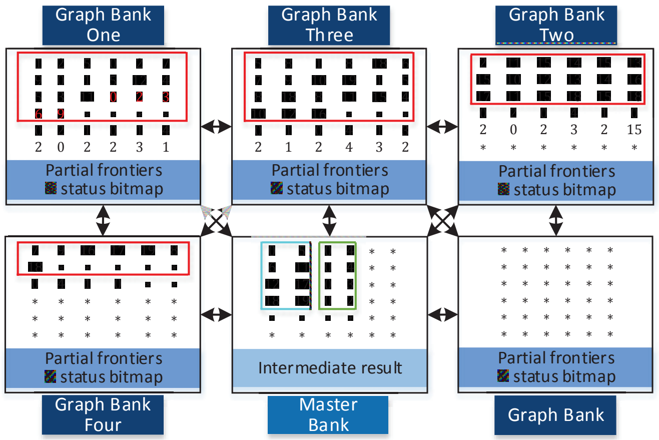

Fig. 4: The layout of graph G involved with multiple banks.

C. *Graph Layout in Multiple Crossbar Arrays*

To efficiently traverse a big graph, multiple ReRAM memory banks should coordinate together. This is well illustrated with the example shown in Figure 3. The adjacency list of graph G are mapped to multiple memory banks.

Figure 4 shows the mapping of graph G involved with multiple memory banks. Due to the symmetrical structure, here we arbitrarily select a memory bank as the master bank.

In this example, the first three banks store adjacency list of six vertices. The fourth graph bank only stores adjacency lists of last two vertices, and the location pointers are stored in third row. The rest rows could store other data. In master bank, the first row data $(0, 5, 0, 4)$ represents that the range of vertex number in first graph bank is from $Vertex\ 0$ to $Vertex\ 5$ (indicated within blue line), and "0" and "4" mean the starting row number of adjacency list and location pointer respectively (indicated within green line). Suppose that this graph traversal starts $Vertex\ 5$, the first graph bank calculates and checks the location pointer of $Vertex\ 5$ and prior $Vertex\ 4$ to determine the exact location of its adjacency list. The next step is to attain the adjacent vertices of $Vertex\ 5$ from the coordinate $[2,\ 2]$ to $[3,\ 1]$ in crossbar by activating wordline No.2 and No.3 and filtering the results of bitlines according to the offset. Since the expansion of frontiers is sequential in our design, each activating wordline operation may attain adjacent vertices of contiguous frontiers. For example, the whole adjacent vertices of $Vertex\ 4$ and extra partial adjacent vertices of $Vertex\ 5$ are attained after activating wordline No.2. The controller caches the extra adjacent vertices for next frontier expansion task if it hits. Moreover, the space of ReRAM memory can be effectively utilized by adopting this dynamic graph data organization. Multiple graphs can share same ReRAM crossbars for parallel traversal.

IV. Breadth-first search on ReRAM-based main memory

In this section, we briefly describe BFS traversal algorithm employed in our RPBFS architecture with accompanying pseudo-code. To implement BFS traversal, there are three stages from data storage to hardware execution: graph mapping, graph initialization and BFS traversal.

A. Graph Mapping

A ReRAM memory bank is selected to be master bank for a graph, and multiple memory banks are used as graph banks to store the adjacency list. After finishing the adjacency list mapping in one graph bank, master bank records the graph banks information including the vertex range information, the starting row number of the adjacency list, and the corresponding locations. If there are some updates to a persistent graph, both related graph banks and master bank should be updated.

B. Graph Initialization

In BFS initialization stage, the status bitmap is initialized with a source vertex. Algorithm 1 and Algorithm 2 give the pseudo-code of initialization stage of master and graph banks respectively. Since master bank is fully interconnected with all the graph banks, it sends the corresponding vertex range information and starting row number to graph banks. Then it creates a vertex status bitmap (FSB) in shared memory, as well as updates it with a source vertex s (line 3). The last step is to record the sum of frontier for next level expansion in order to determine whether the traversal has been finished.

Algorithm 1 BFS in master bank.

```
Input: Source vertex $s$.
BFS initialization:
1: Send the corresponding vertex range and the starting row number to each
   graph bank.
2: Full status bitmap in Shared Memory $FSB \longleftarrow \varnothing$
3: $FSB[s] \longleftarrow 1$
4: The sum of frontiers for next level expansion $frontier\_sum \longleftarrow 1$
BFS traversal:
1: while $frontier\_sum \neq 0$ do
2:     Send $start\_cmd$ to all graph banks
3:     $frontier\_sum \longleftarrow 0$
4:     Waiting for $frontier\_num$, $finish\_cmd$ from each graph bank
5:         for each $frontier\_num$ do
6:             $frontier\_sum \longleftarrow frontier\_sum + frontier\_num$
7:         end for
8: end while
```

Algorithm 2 BFS in graph bank.

```
BFS Initialization:
1: Save the vertex range and the starting row number in register.
2: Partial prior status bitmap in cache $PPSB \longleftarrow \varnothing$
3: Partial status bitmap in cache $PSB \longleftarrow \varnothing$
4: Partial frontier bitmap in cache $PFB \longleftarrow \varnothing$
BFS Traversal:
1: Waiting for $start\_cmd$ from master bank
2: $PSB \longleftarrow FSB$
3: $PFB \longleftarrow PSB \otimes PPSB$
4: The number of frontiers in $PFB$ $frontier\_num \longleftarrow 0$
5: for each $u$ is non-zero in $PFB$ do
6:     $frontier\_num \longleftarrow frontier\_num + 1$
7:     Get location pointer of $u$ in crossbar arrays
8:     Get adjacent vertices $\{A\}$ in crossbar arrays
9:     for each $v$ in $A$ do
10:        $FSB[v] \longleftarrow 1$
11:    end for
12: end for
13: $PPSB \longleftarrow PSB$
14: Output($PFB$)
15: $PFB \longleftarrow \varnothing$
16: Send $frontier\_num$, $finish\_cmd$ to master bank
```

The initialization stage in graph bank is shown in Algorithm 2, each graph bank saves the corresponding vertex range information and starting row number in registers for accelerating expansion. Moreover, according to the vertex range information, each graph bank generates its own empty partial bitmaps for performing a sub-graph traversal procedure (line 2-4).

C. BFS Traversal

In this stage, the master collaborates with graph banks to finish BFS traversal, which guarantees the BFS level synchronization. In Algorithm 1, if the frontier sum for the next level expansion is not equal to zero, the master bank sends $start\_cmd$ to inform all the graph banks to traverse this graph. After resetting the $frontier\_sum$, the master bank waits until it receives the $finish\_cmd$ along with the partial frontier number from all the graph banks (line 4). Pile those partial frontier numbers on the $frontier\_sum$, the master determines whether the graph traversal has been finished.

Graph bank carries out vertex expansion in this stage, as shown in Algorithm 2. The graph bank is listening its command line, once it receives $start\_cmd$, a partial status bitmap PSB is filled with FSB from the shared memory without memory collision. The size and offset of the PSB are determined by the vertex range information. After that, the operation of ($PSB \otimes PPSB$) generates frontiers for the current level (line 3). For example, there are six vertices stored on a graph bank. Assumed that the prior partial status bitmap $PPSB$ is "001101" in preceding level, and the latest partial status bitmap PSB is "101111", then the frontiers for the current level PFB is "100010" after executing ($PSB \otimes PPSB$). For each non-zero bitmap in PFB, it issues one vertex expansion task that involves attaining location pointer and adjacent vertices in crossbar (line 7-8). The number of iterations of expanding adjacent vertices is bounded by the size of PFB. The next step is to update status bitmap FSB with adjacent vertices of the frontiers in the shared memory. Updating this bitmap uses atomic operations since multiple graph banks could set same bits, which is non-break for the graph traversal. After this level expansion, the graph bank replaces $PPFB$ with PSB, and finally sends the frontier number and the $finish\_cmd$ to the master bank. The frontiers in all the graph banks are the output of each level, they can be either transferred to shared memory or flushed to local ReRAM memory for applications use.

1) Traversal Performance Analysis : Our RPBFS architecture involves multiple memory banks, thus we can utilize bank-level parallelism to accelerate graph traversal. In each level expansion, all the corresponding graph banks read the shared memory without the conflict since there is no access overlap. After that, each graph bank does computation to get frontiers with the bound of vertex range, then expands each frontier. The adjacent list of the frontiers are updated in shared memory, and the atomic updates are bounded by the edge number, and the updates overhead can be reduced by employing ILP and merging same operations. The maximum of expansion time is mainly determined by the maximum number of frontiers in all the graph banks. Since the level synchronization cost in this interconnect network architecture is tiny, therefore a graph involved more graph banks can be discovered faster.

2) Extra Vertex Cache: Besides attaining adjacent vertices of a specified vertex, the extra partial or entire adjacent vertices of contiguous frontiers can also be discovered by activating one wordline. Figure 5 presents the workflow of the extra vertex cache. RPBFS caches the extra vertices with adjacent vertices and the sum of them. Starting an expansion task, the controller will check whether the frontier vertex is cached. If it hits, the controller recalculates location pointers. Since RPBFS expands the frontiers in order, so we use the FIFO as cache replacement policy.

V. Experimental Evaluation

In this section, we present the evaluation results of the RPBFS architecture under various experiments.

A. Methodology

We compare our ReRAM-based graph traversal design with the state-of-the-art CPU-based parallel implementation and

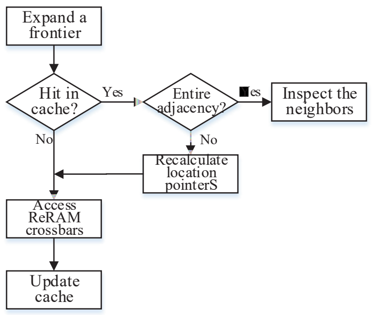

Fig. 5: The workflow of the extra vertex cache.

GPU-based solution Enterprise [4]. RPBFS is modeled by heavily modified NVSim to simulate peripheral circuit [12], as well as the traversal scheme attached by controller which provides control signals to all the peripheral circuits. We modify the simulator as a trace-based system to evaluate performance with other solutions. The related ReRAM and circuits timing parameters are derived from [11] [13].

The configurations of RPBFS architecture and the detailed hardware of other platforms are illustrated in Table I. There are four ReRAM crossbars per graph bank, each crossbar contains 1024 * 1024 ReRAM cells, and we assume that all the cells have the same properties without IR drop effect. The ReRAM cell is assumed as 4-bit MLC. In this part, we use eight cells to represent a vertex, four cells to represent a location pointer.

We perform our tests on five real world workloads for evaluation. The Amazon product co-purchasing network (AM) graph has 400 thousand vertices and 3.2 million edges, Wikipedia talk network (WT) has 2.39 million vertices and 5 million edges. The web graph of Berkeley and Stanford (WB) has more than 7.6 million edges but only 685 thousand vertices. Two related small graphs, email communication network from Enron (ER) only has 36 thousand vertices and 367 thousand edges, and Slashdot social network (SD) with 82 thousand vertices plus 948 thousand edges are also tested. All the graphs are represented by compressed sparse row (CSR) format and mapped to ReRAM memory. We perform sorting operation on these graphs and set the starting vertex number being 0. These pre-processing operations do not change graph topology. All the graph data is loaded into ReRAM-based main memory, DRAM memory and GPU global memory ahead respectively, the timing starts when the source vertex is given and ends when search is completed. We use traversed edges per second (TEPS) to evaluate traversal performance. We firstly show the performance speedup with crossbar scalability. We map the graphs to the ReRAM crossbar with the scale of 256*1024, 512*1024 and 1024*1204 cells respectively. After that, we compare our RPBFS algorithm with other direction-optimizing solutions which have been proved that they perform better than top-down algorithm in traditional platforms.

B. Performance Results

1) Performance Comparison: Figure 6 shows the scalability of RPBFS. When the scale of ReRAM crossbar decreases, more graph banks are involved. For big graphs WT and WB,

TABLE I: The Configurations of RPBFS Architecture and Hardware.

Controller	16 registers; one Core at 1.2GHz
Cache	512KB
Shared Memory	4MB
Internal Bus	50GB/s
ReRAM-based memory	16 Banks/chip; 4 Crossbars/Bank; 1024*1024 Cells/Crossbar; tRCD-tCL-tRP-tWR 18-9.8-0.5-30 (ns)
CPU Cores	Inter Core2 Q9550 with 2.83GHz
CPU L1 Cache	32KB SRAM
CPU L2 Cache	6144KB SRAM
Main Memory	4GB with two channels
Graphics Card	GTX TITAN X with 3072 CUDA cores

the small scale crossbar outperforms the bigger scale up to 2.7× and 2.3×. This is because like WT whose proportion of node number and edge number accounts for around 50%, the frontiers is sufficient in each level so more graph banks perform expansion tasks with the small scale crossbars. While in ER and SD just with several thousands vertices and edges, the performance improvement with small scale crossbars is not obvious due to the low parallelism and the insufficient frontiers.

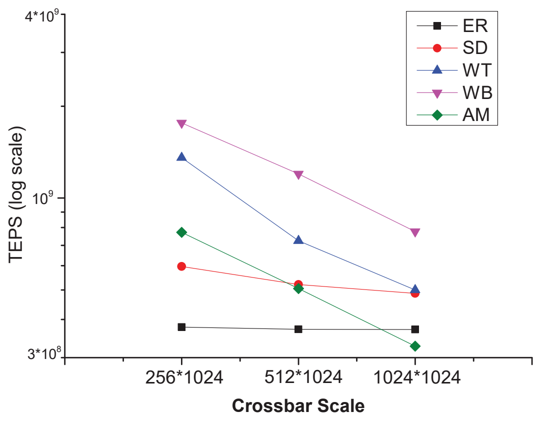

Fig. 6: Performance with the scalability of RPBFS.

Figure 7 compares our RPBFS solution with GPU-based framework Enterprise and CPU-based 64-thread parallelism implementations respectively. RPBFS architecture performs up to 16.0× better than Enterprise, and up to 33.8× better than CPU-based solution. This is because our RPBFS wraps the adjacent list access within memory banks and the data movement in the internal bus is only the vertex bitmap, we reduce the data movement overhead compared with accessing the adjacent list from global memory or main memory.

2) Impact of Extra Vertex Cache: The extra vertex cache technique can accelerate adjacency expansion. We tested different size of extra adjacent lists. The performance improves 3.9% from five entries to ten, while 0.77% from ten to fifteen. The more extra vertex cache can improve hit ratio of continuous frontiers, but it also incurs more vertex ID comparison latency.

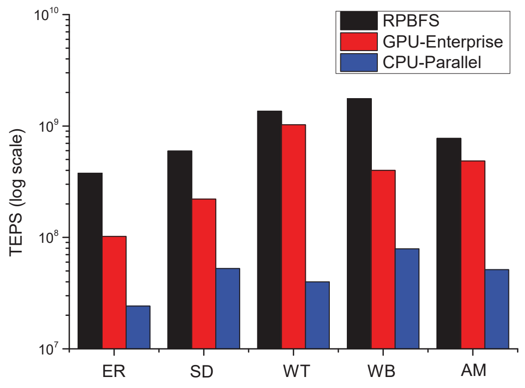

Fig. 7: Performance of RPBFS and direction-optimizing GPU-based and CPU-based solutions.

VI. Conclusion

In this work, we propose a novel ReRAM-based processing-in-memory architecture for breadth-first search, which accelerates big graph traversal by reducing data movement overhead. Benefited from graph distribution in ReRAM crossbars and efficient graph traversal algorithms among memory banks, our architecture can effectively achieve performance improvement compared with other techniques.

VII. Acknowledgments

The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 152138/14E and GRF 152736/16E and GRF 152223/15E), and National Natural Science Foundation of China (Project 61373049), and the Hong Kong Polytechnic University (4-BCBB).

References

[1] F. Alibart and et al., "High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm," in Nanotechnology, 2012.
[2] C. Xu and et al., "Overcoming the challenges of crossbar resistive memory architectures," in HPCA, 2015.
[3] J. Ahn and et al., "A scalable processing-in-memory accelerator for parallel graph processing," in ISCA, 2015.
[4] H. Liu and et al., "Enterprise: Breadth-first graph traversal on gpus," in SC, 2015.
[5] L. Luo and et al., "An effective gpu implementation of breadth-first search," in DCA, 2010.
[6] P. Harish and et al., "Accelerating large graph algorithms on the gpu using cuda," in HiPC, 2007.
[7] S. Beamer and et al., "Locality exists in graph processing: Workload characterization on an ivy bridge server," in IISWC, 2015.
[8] L. Nai and et al., "Graphbig: Understanding graph computing in the context of industrial solutions," in SC, 2015.
[9] M. M. Ozdal and et al., "Energy efficient architecture for graph analytics accelerators," in ISCA, 2016.
[10] S. Beamer and et al., "Direction-optimizing breadth-first search," in Scientific Programming, 2013.
[11] P. Chi and et al., "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in ISCA, 2016.
[12] X. Dong and et al., "Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory," in Emerging Memory Technologies, 2014.
[13] A. Shafiee and et al., "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in ISCA, 2016.