GRAM: Graph Processing in a ReRAM-based Computational Memory

Minxuan Zhou, Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing
Computer Science and Engineering, UC San Diego, La Jolla, CA 92093, USA
{miz087, moimani, sgupta, yek048, tajana}@ucsd.edu

ABSTRACT

The performance of graph processing for real-world graphs is limited by inefficient memory behaviours in traditional systems because of random memory access patterns. Offloading computations to the memory is a promising strategy to overcome such challenges. In this paper, we exploit the resistive memory (ReRAM) based processing-in-memory (PIM) technology to accelerate graph applications. The proposed solution, GRAM, can efficiently executes vertex-centric model, which is widely used in large-scale parallel graph processing programs, in the computational memory. The hardware-software co-design used in GRAM maximizes the computation parallelism while minimizing the number of data movements. Based on our experiments with three important graph kernels on seven real-world graphs, GRAM provides 122.5× and 11.1× speedup compared with an in-memory graph system and optimized multi-threading algorithms running on a multi-core CPU. Compared to a GPU-based graph acceleration library and a recently proposed PIM accelerator, GRAM improves the performance by 7.1× and 3.8× respectively.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Hardware** → *Emerging technologies*;

ACM Reference Format:
Minxuan Zhou, Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. GRAM: Graph Processing in a ReRAM-based Computational Memory. In *ASPDAC '19: 24th Asia and South Pacific Design Automation Conference (ASPDAC '19), January 21–24, 2019, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3287624.3287711

1 INTRODUCTION

Graph is a powerful representation for data collected from different real world domains. With the current trend indicating an explosive growth of data in the near future, processing large graphs in an efficient way, therefore, has become significantly important. However, previous work found that the conventional computer architecture is inefficient when processing large-scale real-world graphs [1–3]. Such inefficiency mainly comes from inconsistencies between conditions favorable to hierarchical memory designs and characteristics of graph processing like low compute-memory ratio and random memory access patterns. Therefore, graph processing applications are usually memory latency- or bandwidth-bound [3, 4].

Processing in-memory (PIM) is a promising approach to solve such memory inefficiencies on many big data applications [5–8].

ASPDAC '19, January 21–24, 2019, Tokyo, Japan

ACM ISBN 978-1-4503-6007-4/19/01…$15.00
https://doi.org/10.1145/3287624.3287711

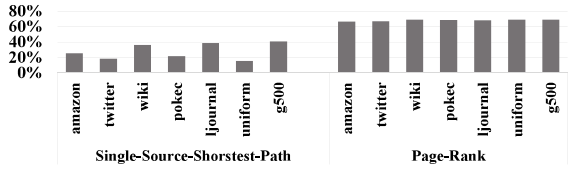


Figure 1: The percentage of execution time spent on data movements the state-of-the-art ReRAM-based graph accelerator [14].

Offloading computations to the memory can provide significant benefits as it reduces the memory access latency and the offchip bandwidth usage. Furthermore, PIM can also provide a high-degree of parallelism to significantly accelerate a large amount of independent computations because all memory cells can be used as processing units. ReRAM is one of the most popular techniques used by various PIM designs, which has shown a great potential in not only high-performance non-volatile memories (NVMs) but also processing units by exploiting its analog or digital characteristics [9–13]. For example, the analog-based ReRAM PIM technology can be used to accelerate different applications by implementing in-memory vector and matrix operations [14–16]. Each ReRAM cell in the crossbar stores a multi-bit value represented by a resistance level. The PIM ReRAM blocks implements different operations by applying specific voltage on rows of ReRAM cells to change their resistance levels. Due to the overhead of converting signals between digital and analog domains, most of the existing analog-based PIM solutions are standalone accelerators [14, 16].

Therefore, these accelerators still have data movement issues because the data is not stored in the computational memory. Fig. 1 shows the execution times spent on data movements in different graph workloads in a state-of-the-art analog-based ReRAM PIM accelerator [14]. In applications like Page Rank, which utilize fast in-memory computations (parallel multiplication-accumulations), data movements may take up to 69% of the total execution time. Furthermore, a previous work [16] shows that ADC and DAC, which are required for signal conversions, consume around 80% power of each ReRAM block which significantly hurts the power efficiency.

Unlike representing data in the analog domain, digital-based ReRAM PIM technology stores two resistance states which represent single-bit values without analog/digital signal conversions. There have been several works utilizing such PIM technologies to implement in-memory computations like bit-wise operations, additions, multiplications and efficient in-memory associative search operations [17–20]. Compared to the analog-based PIM, the digital-based technology is more compatible with existing systems and more power-efficient. These characteristics enable us to build a computational memory which not only serves as a normal memory, but also supports in-place computations. Updating data in-place can significantly reduce the off-chip data movements between the processing units and the memory.

In this work, we propose to accelerate graph processing applications in a computational memory based on the digital-based ReRAM PIM technology. The proposed solution, GRAM, can efficiently execute graph processing algorithms based on vertex-centric

Table 1: Vertex-centric Implementations of Different Algorithms

Application	*vProp*	Process(u, e)	*Reduce*(u, e)	*Apply*
Breadth-First Search	Level	$eProp = u.vProp + 1$	$u.vTemp = min(eProp, u.vTemp)$	$u.vProp = min(u.vProp, u.vTemp)$
Single-Source-Shortest-Path	Distance	$eProp = u.vProp + weight$	$u.vTemp = min(eProp, u.vTemp)$	$u.vProp = min(u.vProp, u.vTemp)$
Page Rank	Page rank score	$eProp = u.vProp * u.out\_degree\_factor$	$u.vTemp = u.vTemp + eProp$	$u.vProp = a * u.vProp + base$

Algorithm 1: Vertex-Centric Model

```
1: for u : ActiveList do                                   ▷ Phase - Process
2:     for v : OutNeighbor(v) do
3:         eProp(u, v) = process(v.vProp, edge(v, u))
4:     end for
5: end for
6: for u : AllVertices do                                  ▷ Phase - Reduce
7:     for v : InNeighbor(v) do
8:         u.vTemp = reduce(u.vTemp, eProp(v, u))
9:     end for
10: end for
11: for v : AllVertices do                                 ▷ Phase - Apply
12:     v.vProp = Apply(v.vProp, v.vTemp)
13: end for
```

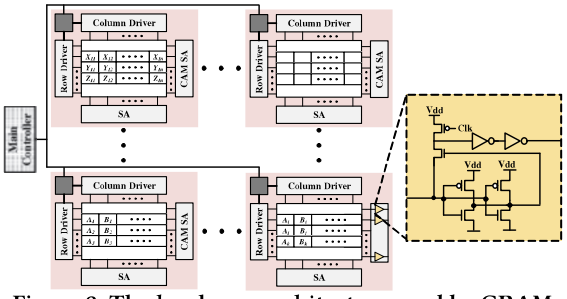


Figure 2: The hardware architecture used by GRAM.

model, which is widely used to implement various parallel graph algorithms based on bulk synchronous model [21, 22], in the PIM architecture. Algorithm 1 shows the vertex-centric model used in this work which divides each super-step into three three phases: `Process`, `Reduce` and `Apply`. Each phase executes a application-specific function for a set of vertices. All three functions are used to calculate application-specific properties related to vertices and edges (`eProp`, `vTemp`, and `vProp` shown in Algorithm 1). A vertex-centric program runs iteratively on a subset of vertices, called `active vertices`, until it converges. The vertex-centric model is flexible to implement a wide range of graph algorithms by defining application-specific functions in three phase. Table 1 shows vertex-centric implementations for three important graph algorithms.

To fully utilize the computational memory, GRAM, organizes vertex-centric program data, including graph structure data and vertex-/edge-related properties, in the memory to enable various PIM operations. We design a PIM processing flow of vertex-centric program based on the specialized data allocation. In order to increase the PIM computation parallelism, we propose a new hardware design to support one key operation, compare-and-swap (CAS), along with leveraging previous published bit-wise operations, additions, multiplications, search [17–20]. We further propose a hardware-software co-design technique based on efficient in-memory associative search operations to parallelize the `Reduce` operations. Both of these two techniques significantly increase the parallelism available during the execution of vertex-centric programs in the computational memory.

We test GRAM for running three important graph kernels on seven real-world graphs. The results show that GRAM can provide 122.5× and 11.1× speedup compared with an in-memory graph system [23] and optimized multi-threading algorithms [24] running on a powerful multi-core CPU. Compared with a GPU-based graph acceleration library [25] and a recently proposed PIM accelerator [14], GRAM improves the performance by 7.1× and 3.8× respectively.

2 GRAM DESIGN

In this section, we introduce the design of GRAM, and the execution of in-memory vertex-centric graph processing programs in the computational memory. We first design the overall architecture used in this work which consists of multiple ReRAM arrays based on the digital-based PIM technology. Then, we design the data organization of the vertex-centric program in the computational memory to maximize the parallelism provided by PIM operations. Based on the architecture and data organization, we introduce the processing flow of vertex-centric graph processing programs in GRAM.

2.1 GRAM Architecture

The architecture used by GRAM is based on a computational memory supporting various types of PIM operations in digital-based ReRAM blocks. Fig. 2 shows the architecture for the computational memory used by GRAM. Each ReRAM block has a light-weight block controller and shares a centralized main controller with the other blocks. A block controller manages the column and row circuitry of a block which further control the wordline and bitline voltages. The main controller handles both normal memory operations and PIM operations by sending commands to appropriate block controllers. In-memory computations can be organized as vector operations which enable parallel computing. Once a block controller receives a command, it can work independently of the other blocks which allows for the block-level parallelism, where all the blocks can process different sub-vectors in parallel. Besides arithmetic operations, the ReRAM block also supports efficient in-memory search operations. To enable search operations, the memory can be organized as look-up tables with multiple data fields where each data field is allocated to a continuous set of columns inside a memory block. An in-memory search operation activates table entries with a target value in a specific data field and the results are stored as a bit-array in content-addressable memory (CAM) sense amplifier (SA). In-memory search operations also exploits the block-level parallelism by searching in different blocks simultaneously.

2.2 In-Memory Data Organization

To achieve the full functionality of PIM architecture, including the massively parallel computing and efficient search operations, data should be organized carefully in GRAM. We then introduce the in-memory data layout when running vertex-centric graph processing programs. Fig. 3 shows an example of a vertex-centric SSSP algorithm running on a simple graph. The basic graph structure is represented by a vertex table and a edge table. The edge table (`ET`) stores information of each edge including source vertex, destination vertex, and weight. The vertex table (`VT`) stores the edge table index of the first outgoing edge from each vertex for edge traversals.

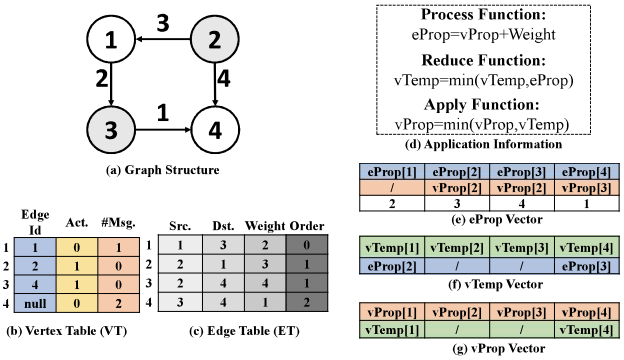


Figure 3: GRAM data organization.

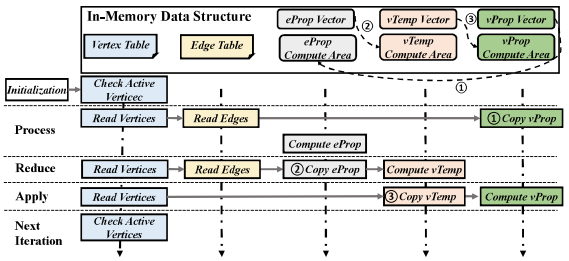


Figure 4: GRAM processing flow.

Three vertex program properties are stored in vectors where `eProp` vector has `M` elements while `vTemp` and `vProp` vectors have `N` elements. `M` and `N` are the number of edges and vertices in the graph respectively. Three vertex program properties are stored as in-memory vectors in one or more ReRAM blocks. Two vectors can be processed simultaneously by utilizing the block-level parallelism. Two tables, `ET` and `VT`, are stored in the computational memory by organizing different fields column-wisely in ReRAM blocks. In-memory search operations can be applied to a specific data field (e.g. destination vertex) to activate target entries. Depending on the application requirement, we can add extra fields in both two look-up table under the constraint of ReRAM row-size. For example, the active flag is required to indicate the active vertices for the current iteration in several vertex-centric graph algorithms. In Section 3.2, we also propose a hardware-software co-optimization which utilizes extra fields in both `ET` and `VT`.

2.3 Processing Flow

We then introduce how GRAM executes an in-memory vertex-centric program in the computational memory. The processing flow is shown as Fig. 4. In the initialization stage, GRAM allocates different application data structures in the computational memory as in-memory vectors and look-up tables. All values, such as initial properties and active vertices, are initialized based on application-specific requirements. In the `Process` phase, the `eProp` vector is updated based on `vProps` of active vertices (shown as grey circle in Fig. 3(a)). Such computations can be processed in parallel because there is no data dependency. The updated `eProp` vector is then used to update the `vTemp` vector in `Reduce` phase. We should note that each `vTemp` may be updated by multiple `eProps` from the vertex's incoming edges. Therefore, computations for a `vTemp` must be serialized in order to correctly update values. The computed `vTemp` vector is then used to updated the `vProp` vector in the `Apply` phase if some vertices has updated `vTemps` (vertex 1 and 4 in the example).

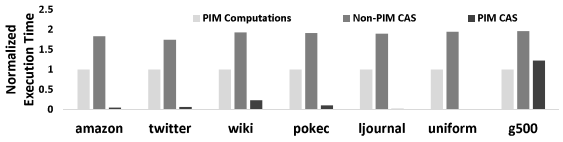


Figure 5: Normalized execution time of parallel PIM computations and two CAS implementations.

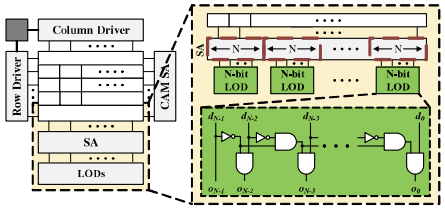


Figure 6: N-bit leading one detector (LOD) circuit used to implement CAS.

The updated `vProp` vector is used to update active vertex list for the next iteration. If there is no active vertex or the program converges, GRAM terminates the current application and returns the result.

3 HARDWARE/SOFTWARE CO-OPTIMIZATION

The basic GRAM design may encounter two challenges when processing vertex-centric graph processing programs. First, the functionality of PIM-enabled ReRAM may introduce a large overhead when executing some operations like compare-and-swap (CAS). Therefore, we propose and design a circuit in the PIM-enable ReRAM block to provide in-memory parallel CAS operations. Furthermore, the `Reduce` phase requires sequential operations because of data dependencies. Such sequential execution significantly slows down the performance of GRAM. We then propose a scheduling mechanism based on in-memory associative search operations to maximize the parallelism in the `Reduce` phase. Both of these two techniques significantly increase the parallelism during the execution of vertex-centric programs.

3.1 In-Memory Compare-and-Swap (CAS)

The limitation of PIM functionality may degrade the parallelism when handling operations with logical predication like CAS operations. CAS is a commonly used operation in many graph applications, which updates the target data based on the comparison between old and new values. CAS computations cannot be naturally handled by the computational memory. Instead, these operations should be processed by a specific ALU present in the main controller which serially reads values (results from previous PIM operations) from the memory and writes the updated values based on comparisons. These serial operations may introduce significant performance overheads. Fig. 5 shows the average ratio of execution time spent on parallel PIM computations and serial CAS computations in breadth-first-search on different graph workloads. We consider the overheads of block-to-controller data movements and computations in both the main controller and the memory blocks. The result shows that the overhead of such comparisons may take up more than 70% of the total execution time.

In order to revolve these inefficiencies, we propose a parallel in-memory CAS operation with a slight modification to the ReRAM block hardware. Such operation updates large vectors without sequentially comparing the results in the main controller. To process

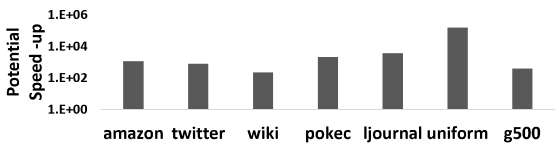


Figure 7: Potential speedup of parallelizing the Reduce phase over sequential execution. The serialized execution of the Reduce may take up to 96.2% execution time in a vertex-centric program.

Algorithm 2: Parallel Reduce

```
1: cur_o = 1
2: while rst = ET.search(Order = cur_o) do
3:     for e : rst do
4:         copy(eProp[e], vTemp[e.dst].compute_row)
5:     end for
6:     reduce_valid(vTemp)
7:     cur_o++
8: end while
```

a CAS operation, the hardware first computes in-memory bitwise XOR of all the pairs present in two vectors and the results are stored in a processing row. It is a single step process since bitwise XOR can happen in parallel for all word pairs. Then, the most significant '1' is searched for in the output of each pair to detect the first bit with mismatch using leading 'one' detector circuits (LODs). These are implemented in the row sense amplifiers as shown in Fig. 6. The inputs ($d_{N-1}, d_{N-2}...d_0$) of N-bit LOD circuit are connected to the outputs of N 1-bit memory SAs, where N is the size of a data word. Each LOD has N output bits ($o_{N-1}, o_{N-2}...o_0$) and only the bit corresponding to the most significant '1' goes high. The leading '1' detection happens in parallel for all word pairs in a block. The bits determined by the output of LODs are read from vector A. For the '1' bits, the corresponding words in B are copied to vector A. This approach replaces the slow sequential out-of-memory comparisons with parallel in-memory computations. This parallel in-memory CAS results in a significant execution time reduction, as shown in Fig. 5.

3.2 Parallel Reduction

The Reduce phase in the vertex program cannot be efficiently processed in the original processing flow. During the Reduce phase, vTemp of each vertex may be updated several times depending on the graph structure. If the PIM architecture runs in the original order, all these computations should be processed sequentially. However, there still exists parallelism in computations for *vTemp*s of different vertices. For instance, if the computation of $vTemp[1]$ requires $eProp[1]$ while the computation of $vTemp[2]$ requires $eProp[2]$ and $eProp[3]$, we can exploit PIM operations to calculate $[vTemp[1], vTemp[2]] = func(eProp[1], eProp[2])$, and $vTemp[2] = func(eProp[3])$ in only two steps instead of three. Serially executing the Reduce may introduce a significant overhead considering the large number of edges in a graph. Fig. 7 shows the potential speedup of parallelizing these computations over executing the original Reduce phase in GRAM.

We then utilize in-memory search operations to schedule parallel operations available in the Reduce phase with the modified look-up tables for graph structure. We add extra data fields in both vertex and edge tables and propose a new programming paradigm to maximize the computation parallelism. Specifically, the "#Msg" field in VT and "Order" field in ET record the statistics for vertices and edges in the Process phase. The "#Msg" field in VT indicates how many in-coming edges of the vertex are processed in the Process phase. The current "Msg" value when processing a specific edge can be used as the order of reduction in the Reduce phase. Thus, we record this number in the "Order" field of ET. When copying a vProp to the computation area of each *eProp*, the "#Msg" of the destination vertex in VT increases by 1 and the new "#Msg" value is used to set the "Order" of corresponding entry in ET.

Algorithm 2 shows the new processing flow in Reduce which utilizes the parallel reduction. During the Reduce phase, we search each "Order" (from 1) in ET and copy target *eProp*s to corresponding computation areas of *vTemp*s. The *eProp*s with the same order number can be processed in parallel because destination vertices of corresponding edges are different. If there is no entry in ET with a specific order number, this process ends and the application continues to the Apply phase. The proposed parallel reduction maximizes the parallelism existing in the Reduce phase by scheduling as many independent computations as possible.

Table 2: Baseline System Configurations

CPU Baseline	
Cores	Intel i7-7700k @ 8 cores 4.5 GHz
Cache	L1: 256KiB, L2: 1MiB, L3: 8MiB
Memory	32GiB DIMM DDR4 @ 2400MHz
GPU Baseline	
Product	GeForce GTX 1080 Ti
# of CUDA cores	3584

Table 3: Graph Workload Summary

Graph	#Vertices	#Edges	Description
amazon (am) [28]	403K	3.4M	Amazon product co-purchasing network
ego-Twitter (tw) [28]	81K	1.8M	Social circles from Twitter
soc-Pokec [28]	1.6M	30.6M	Pokec online social network
wiki-topcats (wiki) [28]	1.8M	28.5M	Wikepedia hyperlinks
ljournal (lj) [29, 30]	5.4M	78M	Live Journal
uniform (u) [24]	2.1M	33M	Uniform random graph
g500 (g) [24, 31]	2.1M	32M	Kronecker graph (Graph500 specifications)

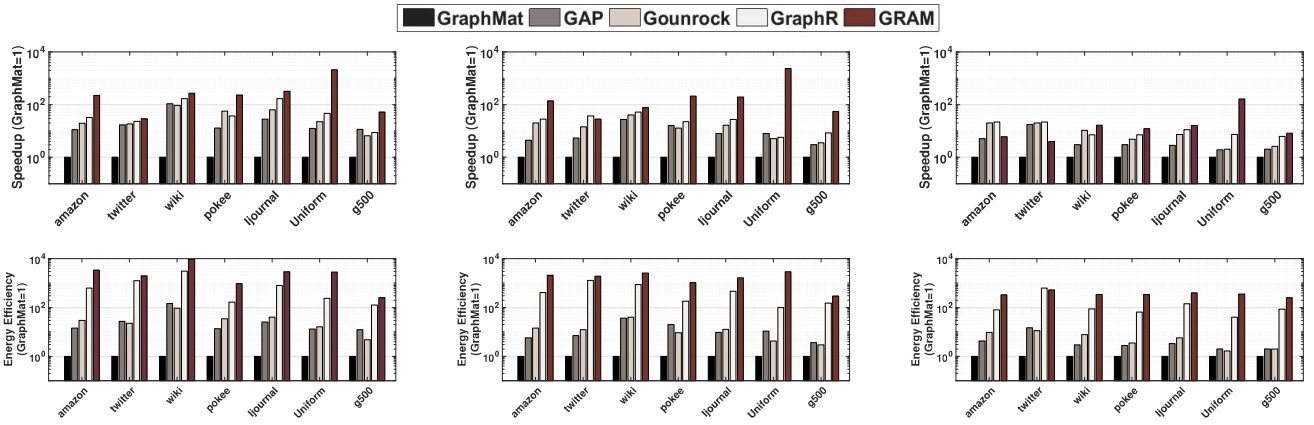
4 RESULTS

4.1 Experiment Setup

We design an in-house simulator to model the detailed hardware functionality of GRAM. All buffers and interconnect are modeled in Cacti [26] at 32nm. For hardware characteristics, we use HSPICE design tool for circuit-level simulations and to calculate energy consumption and performance of all the memory blocks. The energy consumption and performance is also cross-validated using NVSIM [27].

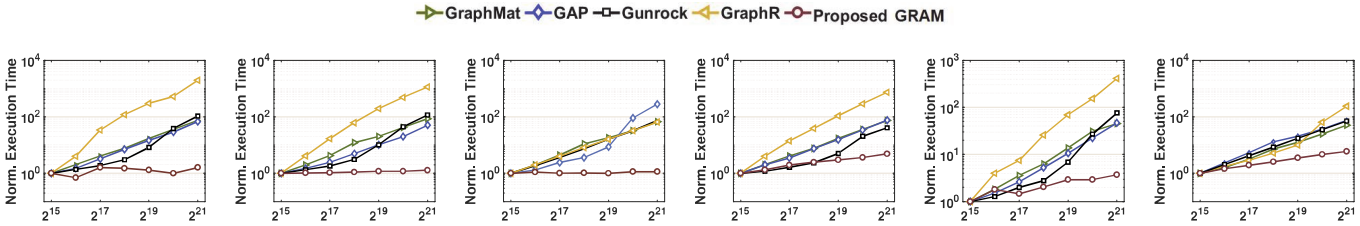
We evaluate 4 different state-of-the-art graph solutions for baseline comparison, including a in-memory graph processing system (GraphMat [23]), an optimized parallel graph benchmark suit (GAP [24]), an NVIDIA GPU accelerated graph library (Gunrock [25]), the most recent PIM accelerator (GraphR [14]). The system configurations are shown in Table 2.

We evaluate three graph algorithms: Breadth-First-Search (BFS), Single-Source-Shortest-Path (SSSP), and Page-Rank (PR) which are the most commonly used graph processing kernels. The application-specific functions based on vertex-centric model has been shown in Table 1. We run all experiments on 5 real world graphs[28–30] and 2 synthetic graphs generated by GAP Benchmark Suite [24, 31]. A summary of tested graphs is listed in Table 3. For the two synthetic graphs, we also generate different numbers of vertices for graph size sensitivity experiments.



(a) Breadth-First Search (b) Single Source Shortest Path (c) Page Rank

Figure 8: The performance and energy efficiency comparison between GraphMat [23], GAP [24], Gunrock [25], GraphR [14] and GRAM. All values are normalized to results of GraphMat.



(a) BFS (uniform) (b) SSSP(uniform) (c) PR (uniform) (d) BFS (g500) (e) SSSP (g500) (f) PR (g500)

Figure 9: The execution time of all systems on uniform and g500 graph with different number of vertices. For each system, all values are normalized to the execution time of that system when executing graphs with 2^{15} vertices.

4.2 Overall Performance

Fig. 8 shows the performance and energy-efficiency results of GRAM, GAP, GraphMat, Gunrock and GraphR over all benchmarks. All results are normalized to values of GraphMat on CPU system.

4.2.1 Execution Time. For performance benefit of GRAM, the geometric means of speedup over CPU-based graph processing framework system (GraphMat) are 204.7× (BFS), 148.9× (SSSP), and 14.0× (PR). Compared with the optimized multi-threaded algorithms (GAP), GRAM can improve the performance by 10.9×, 18.6×, and 3.8× which are also reported as geometric means. These results show that GRAM improves the performance of BFS and SSSP significantly; however, the performance improvement of GRAM over GraphMat on PR is about 1 magnitude less than improvements on BFS and SSSP. The reason is in-memory computation multiplication is much less efficient than conventional CMOS-based ALU. At the same time, GRAM improves the performance of Gunrock by 7.0×, 12.3×, and 2.0× in BFS, SSSP, and PR respectively. When compared to GraphR, the speedups of GRAM are 2.9× (BFS), 7.1× (SSSP), and 1.4× (PR). In general, GRAM outperforms both of these two highly parallel baselines significantly.

4.2.2 Energy Consumption. Fig. 8 also shows the energy consumption results of all solutions. Specifically, average energy-efficiency improvements provided by GRAM over GraphMat are 1975× (BFS), 1469× (SSSP), and 356× (PR). Such great energy savings result from energy-efficient in-memory operations and the fast execution. Unlike the conventional CPU system, which requires frequent data movements between processing cores and the memory, GRAM completes most computation directly inside memory cells and move results to next locations by internal memory buses. Compared to GAP and Gunrock, GRAM reduces the total energy consumption by 112× and 100× respectively. These results show GRAM can improve the energy consumption of graph process on various conventional systems significantly. GRAM also consumes less energy than previous PIM accelerator. Specifically, the energy-efficiency improvements on BFS, SSSP, and PR are 3.84×, 4.24× and 3.38× respectively. These improvements come from both much less data movements and more energy-efficient computations without ADC and DAC logics for multi-level resistance ReRAM cells.

4.3 Scalability

The design of GRAM tries to maximize the parallelism existing in vertex-centric graph processing programs. In this section, we show how the graph size influences the performance of GRAM. Since we cannot scale real-world graphs up, we test GRAM, GraphMat, GAP, Gunrock, and GraphR on two synthetic graphs with different numbers of vertices. Fig. 9 shows the scalability results of all tested solutions where the size of the graph is represented as the total number of vertices. The largest graph in this experiment has 64×

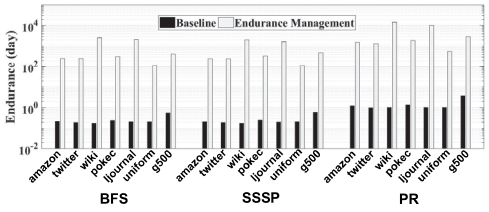


Figure 10: The lifetime extension provided by endurance management.

more vertices than the smallest one. Based on the result, the performance of both GAP and GraphMat increase linearly with the size of the graph, which are 61× and 65× respectively. The execution time of Gunrock increases slightly faster, which is 142×.

GraphR shows the worst scalability in BFS and SSSP, which increases about 700×. This is due to GraphR uses "Add-Op" for these two algorithms which requires a sequential calculation for each row in matrix-vector multiplication on adjacency matrices of subgraphs. The dimension of each subgraph is is $V \times V$ where V is the total number of vertices. On the contrary, the execution time of GRAM increases only up to 4× when the graph becomes 32× larger. This result proves GRAM can provide a good scalability with the graph size which makes it promising for large graph processing problems.

4.4 Endurance Feasibility

Since the NVM device has a limited number of write operations, we then design an two-level (row-level and block-level) round-robin mechanism to improve the endurance of GRAM. The endurance management is triggered offline and changes the address mapping inside ReRAM blocks. We assume ReRAM technology can support 10^{11} write/erase cycles in total [32]. We records number of writes happening in each memory location during one execution of all benchmarks. Estimated lifetime is calculated based on execution time and the most heavily written memory cells. Fig. 10 shows the result of lifetime estimation for each benchmark running with and without our proposed endurance management. Based on these results, the busiest block may fail in less than 1 day without any endurance management because of very uneven write distribution. Our proposed endurance management mechanism can extend the lifetime of GRAM to 469 days, 446 days, and 2640 days in average for BFS, SSSP, and PR respectively. Considering facts that NVM is usually cheaper than other memory technologies and it's unlikely to run applications repeatedly without any spare time, such endurance improvement makes GRAM a feasible NVM-based solution for graph processing applications.

5 CONCLUSION

In this work, we propose GRAM to accelerate graph processing applications in a computational memory based digital-based ReRAM technology. GRAM efficiently execute vertex-centric graph processing programs by maximizing the computation parallelism and minimizing the number of data movements. The experiment results shows GRAM can significantly improve the performance and energy efficiency of graph processing applications over various state-of-the-art solutions. Furthermore, GRAM also shows good scalability and feasibility which make it promising for implementing future computing systems.

ACKNOWLEDGEMENTS

This work was partially supported by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and also NSF grants #1730158 and #1527034.

REFERENCES

[1] S. Beamer *et al.*, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *IEEE IISWC*, 2015.
[2] F. Imani *et al.*, "Nested gaussian process modeling for high-dimensional data imputation in healthcare systems," in *IISE 2018 Conference & Expo, Orlando, FL, May*, pp. 19–22, 2018.
[3] T. J. Ham *et al.*, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *IEEE/ACM Micro*, 2016.
[4] M. Zhou *et al.*, "Gas: A heterogeneous memory architecture for graph processing," in *Proceedings of the International Symposium on Low Power Electronics and Design*, p. 27, ACM, 2018.
[5] M. Imani *et al.*, "Rapidnn: In-memory deep neural network acceleration framework," *arXiv preprint arXiv:1806.05794*, 2018.
[6] M. Imani and otehrs, "Exploring hyperdimensional associative memory," in *2017 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 445–456, IEEE, 2017.
[7] M. Imani *et al.*, "Nvquery: Efficient query processing in non-volatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.
[8] M. Imani, S. Gupta, A. Arredondo, and T. Rosing, "Efficient query processing in crossbar memory," in *Low Power Electronics and Design (ISLPED, 2017 IEEE/ACM International Symposium on*, pp. 1–6, IEEE, 2017.
[9] J. Borghetti *et al.*, "'Memristive' switches enable 'stateful' logic operations via material implication," *Nature*, 2010.
[10] S. Kvatinsky *et al.*, "Memristor-based material implication (IMPLY) logic: design principles and methodologies," *TVLSI*, 2014.
[11] Y. Kim *et al.*, "Orchard: Visual object recognition accelerator based on approximate in-memory processing," in *Proceedings of the 36th International Conference on Computer-Aided Design*, pp. 25–32, IEEE Press, 2017.
[12] S. Gupta *et al.*, "Felix: Fast and energy-efficient logic in memory," in *ICCAD*, pp. 25–32, IEEE, 2018.
[13] M. Imani *et al.*, "Ultra-efficient processing in-memory for data intensive applications," in *ACM DAC*, 2017.
[14] L. Song *et al.*, "Graphr: Accelerating graph processing using reram," in *IEEE HPCA*, 2018.
[15] Y. Kim *et al.*, "Image recognition accelerator design using in-memory processing," *IEEE Micro*, 2018.
[16] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, 2016.
[17] S. Kvatinsky *et al.*, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2015.
[18] N. Talati *et al.*, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, 2016.
[19] A. Haj-Ali *et al.*, "Efficient algorithms for in-memory fixed point multiplication using magic," in *IEEE ISCAS*, 2018.
[20] M. Imani *et al.*, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *IEEE ASP-DAC*, 2017.
[21] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, 1990.
[22] G. Malewicz *et al.*, "Pregel: a system for large-scale graph processing," in *ACM SIGMOD*, 2010.
[23] N. Sundaram *et al.*, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, 2015.
[24] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.
[25] Y. Wang, , *et al.*, "Gunrock: Gpu graph analytics," *ACM TOPC*, 2017.
[26] N. Muralimanohar *et al.*, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *IEEE/ACM Micro*, 2007.
[27] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory," in *Emerging Memory Technologies*, Springer, 2014.
[28] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.
[29] P. Boldi *et al.*, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *ACM WWW* (S. Srinivasan *et al.*, eds.), ACM Press, 2011.
[30] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *ACM WWW*, (Manhattan, USA), ACM Press, 2004.
[31] "Graph500 benchmark." http://graph500.org/.
[32] C. Xu *et al.*, "Overcoming the challenges of crossbar resistive memory architectures," in *IEEE HPCA*, 2015.