GraphIte: Accelerating Iterative Graph Algorithms on ReRAM Architectures via Approximate Computing

Dwaipayan Choudhury
School of EECS
Washington State University
Pullman, USA
dwaipayan.choudhury@wsu.edu

Ananth Kalyanaraman
School of EECS
Washington State University
Pullman, USA
ananth@wsu.edu

Partha Pande
School of EECS
Washington State University
Pullman, USA
pande@wsu.edu

***Abstract*—ReRAM-based Processing-in-Memory (PIM) offers a promising paradigm for computing near data, making it an attractive platform of choice for graph applications that suffer from sparsity and irregular memory access. However, the performance of ReRAM-based graph accelerators is limited by two key challenges – significant storage requirements (particularly due to wasted zero cell storage of a graph's adjacency matrix), and significant amount of on-chip traffic between ReRAM-based processing elements. In this paper we present, GraphIte, an approximate computing-based framework for accelerating iterative graph applications on ReRAM-based architectures. GraphIte uses sparsification and approximate updates to achieve significant reductions in ReRAM storage and data movement. Our experiments on PageRank and community detection show that our proposed architecture outperforms a state-of-the-art ReRAM-based graph accelerator by up to 83.4% reduction in execution time while consuming up to 87.9% less energy for a range of graph inputs and workloads.**

***Index Terms* — Processing-in-Memory, Approximate Computing, Graph Analytics, ReRAM.**

I. INTRODUCTION

Graph analytics has become part of machine learning toolkits to analyze relational data in many real-world applications. Considering poor data locality in most of the real-world graphs, irregular data access patterns become a bottleneck in the performance of conventional manycore architectures (such as CPUs and GPUs). Moreover, skewed vertex degree distributions of real-world graphs cause repeated accesses to vertex neighborhoods or random walk traversals to incur a high volume of cache misses.

Resistive random-access memory (ReRAM)-based Processing-in-Memory (PIM) modules offer an effective way to address the high memory bandwidth requirement of graph analytics by integrating the computing logic in the memory. To perform graph computations on ReRAM crossbars, it is necessary to load the input graph as an adjacency matrix so that the underlying primitives can be decomposed into multiply-and-accumulate (MAC) operations. However, large graph sizes in the real-world (with millions of rows implying trillions of cells) make it prohibitive to load or store the entire adjacency matrix. It is also rather unnecessary to do so because most real-world graphs tend to be highly *sparse*, with the number of nonzero entries orders of magnitude fewer than the number of cells. Graph computations usually only use the nonzero entries. Sparsity also affect locality since the nonzero cells may not be necessarily contiguous in the input matrix. Subsequently, the question arises on how to store a large sparse matrix on an ReRAM platform without wasting space *and* without compromising on performance or energy benefits.

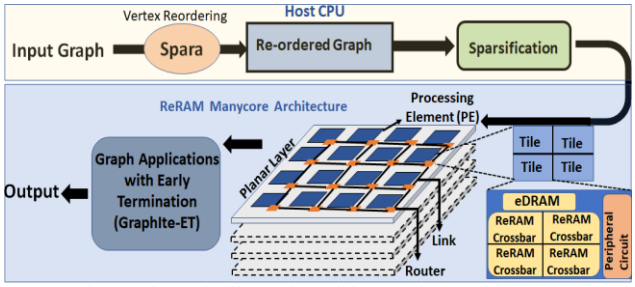

Fig 1: Schematic illustration of the GraphIte architecture.

Contributions: In this paper, we design approximate computing techniques for executing iterative graph applications on ReRAM-based architectures. We refer to our proposed approach as *GraphIte* (Fig. 1 shows a schematic illustration). Approximate computing [2] is a broad class of techniques that use heuristic schemes to achieve the best of performance-precision tradeoffs in real-world applications. GraphIte uses two types of approximate computing techniques as follows:

Sparsification: First, we present a *graph sparsification approach* to selectively determine and eliminate large portions of the adjacency matrix dominated by zero entries (sparse tiles), while retaining parts that are concentrated with non-zeros (dense tiles). This approach helps not only in significant reductions in ReRAM storage, but it also improves the achievable performance and energy efficiency.

Approximate update: Next, we present an *approximate update* method by which vertices are selectively and dynamically pruned (or terminated) as the algorithm proceeds on the ReRAM in iterative steps. This is a generic technique that can be applied to any graph operation with an iterative structure, where all vertices are visited at each iteration (e.g., PageRank, community detection) [3]. A higher level of pruning corresponds to larger savings in time (and data movement), however with the potential risk of degrading quality. Therefore, a careful design is necessary to make this idea work in practice for real-world graph applications.

We implemented the above two types of approximate computing techniques for two different graph operations – namely, PageRank [4] and community detection [5]. Both these operations are exemplars of iterative graph methods that iterate repeatedly over all the vertices until a point of convergence.

We perform a thorough experimental evaluation of the GraphIte-based implementations of PageRank and community detection on an ReRAM-based architecture with 1,024 processing elements (PEs) connected using a Network-on-Chip (NoC) architecture. Results show that the GraphIte implementations are highly effective in reducing storage requirement, time to solution as well as energy costs – all

978-3-9819263-7-8/DATE23/© 2023 EDAA

without compromising the output quality. GraphIte with sparsification and early termination, called GraphIte-ET outperforms a state-of-the-art ReRAM-based design by up to 83.4% reduction in execution time while consuming up to 87.9% less energy for a range of graph inputs and workloads.

II. Related Work

Due to irregular memory accesses in most of the real-world graph applications, data movement between logic and memory layer limits the performance and energy efficiency of CPU and GPU-based conventional manycore architectures. DRAM-based Hybrid Memory Cube (HMC) is an effective way to improve performance by closely integrating the memory with the logic layer [19]. Another possible way is to partition the caches into multiple planar layers in a 3D structure to improve the cache hit rate [20]. However, such deep memory hierarchies also degrade the overall performance. Alternatively, due to in-memory processing capability, ReRAM-based architectures are gaining momentum as a natural choice for accelerating graph operations [6][7][8]. These accelerators outperform CPU- or GPU-based implementations in terms of execution time and energy [6]. While reliability due to hardware faults is a well-documented problem with ReRAM platforms, a number of fault-tolerant schemes being proposed (such as error-correction codes (ECC) [10], redundancy [11]) that enable reliable operation on ReRAMs. Therefore, in this paper, we primarily focus on improving the performance and energy efficiency of graph processing on reliable ReRAM architectures.

Performance of the current ReRAM-based accelerators is limited by the sparsity and lack of locality in graph structures [7][8]. Two recently proposed ReRAM-based graph accelerators (GraphSAR [7] and Spara [8]) leveraged vertex reordering techniques to improve the sparsity-induced inefficiency. While vertex reordering can help by clustering the non-zero cells of the matrix, new algorithmic strategies to fully exploit the reordered structure are needed to realize performance and storage benefits on ReRAMs. More specifically, reordering can rearrange the nonzeroes in the matrix in such a way that there is a clearer separation between denser and sparser "tiles" (or submatrix blocks). The schemes presented in this paper takes advantage of this observation.

Approximate computing [2] generally works by trading off quality for performance. The main idea is to find ways to skip portions of computation such that the overall quality of the solutions is not significantly perturbed while enhancing the performance and energy efficiency [3]. One challenge in implementing approximate computing for ReRAM-based platforms arises from the adjacency matrix-based representation to load and compute on the graph (compared to more traditional formats like adjacency/edge lists or compressed sparse row). Another challenge arises owing to the crossbar structure of ReRAMs. In this paper we tackle both these challenges.

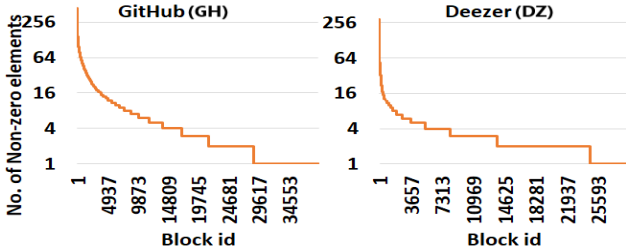

Fig. 2: No. of non-zero elements in each block for GH and DZ, after Spara.

III. Approximate Computing on ReRAM

A. Identifying active blocks using Sparsification

Graph computations on ReRAM-based architectures involve traversing the input sparse adjacency matrix corresponding to the graph. For a graph $G(V,E)$ with n vertices, the corresponding adjacency matrix has n^2 cells. However, most of the real-world graphs are sparse in nature with orders of magnitude fewer nonzero values (i.e., edges) than n^2. Therefore, storing the entire adjacency matrix will be wasteful and prohibitive in practice.

Here, we present a sparsification based approach toward reducing the storage requirement on ReRAMs. To accomplish this reduction, we first define the term active block. A square tile of a matrix of size X rows * X columns is considered "active" if it contains at least one cell with a nonzero value. Since graph computations only involve the nonzero cells of a matrix, we need to transfer only the active blocks of the adjacency matrix onto the ReRAM. A simple but naïve decomposition of the input adjacency matrix into evenly sized active blocks may not necessarily reduce storage in practice as nonzeros can be scattered across the matrix. To this end, vertex reordering techniques can be used [7]. Intuitively, the idea is to reorder the rows and columns of the matrix in such a way that the nonzeros are clustered along the main diagonal [7][8]. As part of this work, we used the Spara reordering [8] although any vertex reordering of choice can be used.

However, even after reordering, there may be several blocks which are highly sparse. Fig. 2 shows the distribution of the number of nonzero cells within each active block for two real-world graph datasets (GitHub and Deezer) after reordering using Spara (using a crossbar size of 128 x128 as an example). Fig. 2 shows a skewed distribution where most of the blocks have very low number of nonzeros (i.e., still very sparse). If one were to store all active blocks on the ReRAM, then that will result in substantial wasted space devoted to storing zero cells.

To reduce the number of active blocks after vertex reordering, we use sparsification, which removes a subset of edges (i.e., nonzero cells). Unlike conventional schemes that remove edges randomly, our sparsification approach prioritizes removal of the sparser active blocks until the desired level of sparsification is achieved. We use a parameter called sparsification factor (SF) that denotes an upper limit on the fraction of edges to be removed prior to loading the graph onto the ReRAM. To maximize the number of active blocks that can be eliminated, we process the active blocks in the decreasing order of their sparsity, until a total of SF fraction of edges is removed. Note that sparsity of a block is simply the fraction of cells that has nonzero entries. In Section IV, we show that choosing an optimized value for SF causes insignificant precision loss while significantly reducing the storage requirement, execution time and energy.

B. Approximate Updates for Iterative Graph Algorithms

In the next step, we describe an approximate update scheme that is performed *during* the computation stage once the graph is loaded on the ReRAM platform. Our technique

applies to any graph algorithm that has the following iterative structure:

1) Initialize a state (or value) at every vertex.
2) Perform multiple iterations such that at each iteration the states (or values) of each vertex is updated using the states (or values) of its neighboring vertices.
3) The algorithm terminates when a convergence criterion is achieved.

The above computational structure is generic and applies to a broad class of iterative graph algorithms. For instance:

- In the *PageRank* algorithm [4], the value computed at each vertex *u* is its PageRank value, which is updated at each iteration using the PageRank values of *u*'s neighbors.
- In *community detection* [5], the state computed at each vertex *u* is its community label. Two actions are possible for *u* at each step – either *u* stays in its current community, or it leaves its current community and joins one of the communities of its neighbors. This greedy decision is made based on whichever action maximizes net gain in modularity [3] – a global objective function.
- In balanced *distance-1 coloring* [2], the state of each vertex represents its color, which is updated at each iteration based on the colors used by its neighbors.
- In the Bellman-Ford s*ingle source shortest path* algorithm [18], the value updated at each vertex *u* is its most up-to-date shortest path distance from the source.

In all the above iterative graph algorithms, the graph algorithms progress toward convergence at each step of the iteration. Subsequently, most of these algorithms show a diminishing returns property [3], whereby the returns in the improvement of quality diminishes with every passing iteration. This happens, however, *without* any reduction in the work performed as all vertices are processed at each iteration. This is the key property that we exploit in this paper to design our approximate update method. Our scheme tries to reduce the work performed at each iteration (adaptively) as the iterations progress. The challenge is to design a scheme which would achieve significant reductions in work without compromising or negatively impacting the output quality.

Next, we describe two such approximate update schemes: one for community detection and another for PageRank. Similar strategies can be designed for other iterative graph algorithms following the template laid out here.

1) Approximate Update for Community Detection

We devise a probabilistic scheme by which a vertex decides to stay "active" or get "terminated", at any given iteration. Being active implies that the vertex will compute its community affiliation and decide whether to change the corresponding community or not, by examining its neighborhood. Alternatively, if the vertex is terminated, it will be dropped from the processing queue during that iteration. Note that by terminating a vertex during an iteration, we can save on all the subsequent computations and inter-PE communication that originate at that vertex. At the start of the first iteration, all vertices are active. As the algorithm progresses through its iterations, more and more vertices will get terminated. Compare this with the baseline (precise) algorithm [5] where all vertices stay active across all the iterations. Consequently, we refer to this approximate update scheme as Early Termination (ET).

To identify vertices to terminate, we track the most recent activity at each vertex – i.e., intuitively, if the community of a vertex has not changed in the past few consecutive iterations, the probability of that vertex staying active is reduced. Specifically, given vertex *v* and its community $C_{v,j}$ at the end of iteration *j*, we assign the probability that *v* is active during iteration *k*, denoted by $P_{v,k}$, as follows [16]:

$$P_{v,k} = \begin{cases} 0, & if\, C_{v,k-3} = C_{v,k-2} = C_{v,k-1} \\ 1, & \text{otherwise} \end{cases} \quad (1)$$

For implementation, a binary flag is used at every vertex to determine the active state of a vertex. This flag is determined based on the probability $P_{v,k}$. If a vertex becomes inactive at a certain iteration, it is not considered as part of future iterations (which implies its community status will no longer be updated). Note that this deviation from precise update may potentially affect output quality. Precision in quality is measured using the modularity metric. This heuristic has two performance advantages: a) it could lead to a faster convergence of modularity within a phase, by reducing both the number of vertices that need to be processed at every iteration and the total number of iterations required, and b) it could also reduce inter-PE traffic generated by terminated vertices.

2) Approximate Update for PageRank

PageRank [4] computes a ranking of webpages (i.e., nodes on a web graph), with a higher value of PageRank denoting more importance to that webpage. The conventional implementation of PageRank is based on the fact that an average web surfer visits page to page, either using the outgoing links of a page (vertex) chosen uniformly at random with probability *d*, or by randomly jumping to a new page (with probability *1- d*). The output of PageRank is a score for each page on the web that determines its importance. The PageRank of a vertex depends on the PageRank of its neighboring vertices. More specifically, consider a directed graph *G(V, E)* with vertex set *V* and edge set *E*. For a given vertex v_i, let $I(v_i)$ be the set of vertex neighbors with incoming links to v_i. The PageRank score for vertex v_i is defined by the equation:

$$PR(V_i) = \frac{1-d}{|V|} + d * \sum_{j \in I(v_i)} PR(v_j) \quad (2)$$

We start by initializing all vertices to an initial PageRank (PR) score of $\frac{1}{|V|}$. PageRank iteratively computes the *PR* value of each vertex using (2) until convergence.

It has been observed that the magnitude of changes in the *PR* values tend to diminish as iterations progress [3]. We exploit this property to retire or early-terminate a source vertex v_i if that change in that vertex' PR value between consecutive iterations drops below a threshold α. More specifically, based on the change in individual PR value of a vertex at a given iteration ($PR(v_i)_k$), we introduce a probability function ($\mathfrak{p}_{v,k}$) which determines vertices that needs to be terminated. This probability during iteration *k* is given by:

$$\mathfrak{p}_{v,k} = \begin{cases} 0, & if |PR(V_i)_{k-1} - PR(V_i)_{k-2}| < \alpha \\ 1, & otherwise \end{cases} \quad (3)$$

Here, α is an input parameter that sets the minimum threshold of PR difference between the previous two iterations to keep a vertex active in the current iteration.

3) Overall ReRAM-based Architecture

In ReRAM-based accelerators, the adjacency matrix of the input graph is stored across the ReRAM cells. During execution, graph computations are decomposed into a set of MAC operations that are performed based on Ohm's and Kirchhoff's current laws. The overall system consists of multiple ReRAM processing elements (PEs), where each PE contains several ReRAM tiles. Each ReRAM tile is composed of several crossbars and the associated peripherals.

It should be noted that both the vertex reordering and sparsification steps are one-time preprocessing steps that are executed on the host machine, and it is only the resulting reordered sparsified graph (with only its identified active blocks) that are loaded on to the ReRAM manycore architecture. For reordering we use the state-of-the-art Spara reordering scheme [8]. Sparsification was described in Section III.A. The approximate update schemes (described in Section III.B) are executed on the ReRAM manycore architecture during the subsequent graph computation phase. Fig. 1 illustrates the overall workflow proposed in this work. The manycore ReRAM architecture with its components is shown for illustration purpose only.

IV. Experimental Results

A. Experimental Setup

For our experimental evaluation, we implemented two different versions of *GraphIte*: the baseline version that uses Spara for graph reordering, followed by our block-based sparsification described in III.A; and an extended version, *GraphIte-ET*, that in addition uses the early termination heuristic described in Section III.B. We modified the open source Grappolo toolkit for the GraphIte implementations with the approximate computing techniques [17].

In the GraphIte architecture, each PE has four tiles. Each tile contains 96 crossbars (128x128) and associated peripheral circuits such as ADC, DAC, etc. Each PE takes up 0.37 mm^2 of area [14]. We consider a 3D architecture to offer a higher degree of integration of ReRAM PEs than the 2D counterparts [9]. By considering 10mmx10mm as the size of each planner layers, such layer contains 256 PEs. Considering four of such planner layers connected on top of each other, it gives rise to a 3D ReRAM-based system with 1024 PEs. Due to simplicity and ease for implementation, we choose a conventional 3D Mesh-based NoC to connect the PEs. Within each layer 256 PEs are placed in a 16x16 grid pattern, and the length of each inter-router link is 0.625mm. We leverage Booksim [13] for implementing 3D Mesh-based NoC architecture considered in this work. The overall system runs at the clock frequency of 2.5 GHz. Considering this clock frequency, each inter-router planar link can be traversed in one cycle. All the vertical links connecting the planar layers are traversed in one cycle due to their small length. BookSim determines the overall NoC latency. We use the PE and memory characteristics along with total NoC latency in NVSim [12] to determine the overall energy consumption and execution time. Table 1 shows all the graph inputs used for the performance analysis. These graph datasets are taken from the Stanford Network Analysis Platform[1] and the Network Repository[2].

Table 1: Input statistics of the graph datasets used in our experiments.

Input graph (label)	No. vertices	No. edges
musae_Github (GH)	37,699	289,003
gemsec-Deezer (DZ)	41,773	125,826
ego-Twitter (TW)	81,306	1,768,149
road_luxembourg-osm (RM)	114,598	119,667
Web-Standford (WS)	281,903	2,312,497
com-Amazon (AZ)	334,863	925,872
roadNet-PA (PA)	1,088,092	3,083,796
Wiki-topcats (TP)	1,791,489	28,511,807
roadNet-CA	1,965,206	5,533,214
com-Orkut (OR)	2,937,612	20,959,854
socfb-A-anon (FB)	3,097,165	23,667,394
soc-LiveJournal1 (LJ)	4,847,571	68,993,773

[1]http://snap.stanford.edu/; [2]http://networkrepository.com/

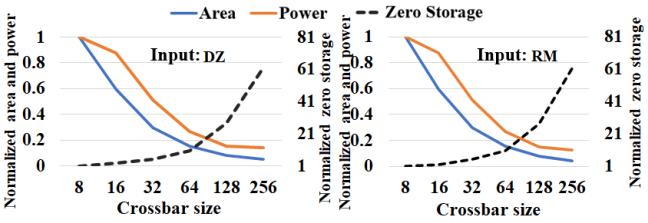

Fig. 3: Area-Power-Zero storage trade-offs for different crossbar configurations. Crossbar size X*X is denoted as X. All values are normalized relative to that of the 8×8 crossbar configuration.

B. Effect of Crossbar Size on Area, Power, and Storage

The adjacency matrix of a graph is decomposed into multiple non-overlapping $N \times N$ segments to map on to $N \times N$ shaped crossbars. Intuitively, selecting relatively smaller crossbars would reduce zero cells stored but also would negatively impact the area and power requirements as those terms are dominated by peripheral circuits [14]. On the other hand, a large crossbar size would reduce area and power but would also potentially increase zero cell storage. We evaluate this tradeoff with multiple inputs. Fig. 3 shows the normalized area, power and zero storage by varying the crossbar size from 8×8 to 256×256. All values are normalized relative to the respective numbers observed for the 8×8 crossbar configuration. While we tested for several inputs, the observed trends were similar and therefore we show the results for only two exemplar inputs. We can see that the area and power continuously decrease with increasing crossbar size. However, beyond 128×128 both area and power show saturating trends, while the zero storage significantly increases (more than 30x over the 8×8 configuration). Hence, we select the 128×128 crossbar size as the default for all our experiments. In this configuration, on average, the area and power are 92% and 85% less than that of the 8×8 crossbar respectively, while the zero wastage is reduced by more than 27.4X.

C. Effect of Sparsification Factor on Quality and Storage

As shown in Fig. 2, active blocks in most real-world graph datasets have varying sparsity. Active blocks with high sparsity not only increase storage requirement but also generate inter-PE traffic. Through sparsity-based approximation, we remove edges belonging to the blocks

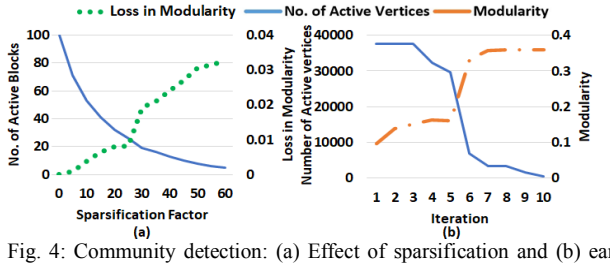

Fig. 4: Community detection: (a) Effect of sparsification and (b) early termination on the normalized number of active blocks (y-axis left) and on the quality of output (precision loss and modularity by the y-axis right).

with comparatively high sparsity. The parameter sparsification factor (SF) denotes the fraction of edges to be removed (relative to the original number of edges in the input graph). We study the number of active blocks by varying the value of SF. As removal of edges may also potentially degrade the precision of the output, we also analyze the loss in precision along with the number of active blocks.

Fig. 4 (a) shows the effect of sparsification on the normalized number of active blocks (y-axis left) and on the precision loss in community detection (y-axis right) with GH. The precision loss in community detection is measured as the difference between the output modularity values calculated by the sparsified implementation versus the implementation without sparsification. We show the result with GH as an example (the same trend is observed for all the others). The number of active blocks with the SF value of zero is 100%.

It should be noted that increasing SF decreases the number of active blocks (i.e., generating more space savings) but it also increases precision loss. While this storage-precision tradeoff is expected, it should be noted that for most of the graph datasets considered in this work, when the value of SF is ≤ 25, the modularity (precision) loss is maintained below 1%, which is desirable from an application standpoint [15]. For this setting, we can achieve 61% to 89% reduction in the number of active blocks, which is a significant savings in space. We use SF value of 25 for all the experiments.

D. Performance with Early Termination (GraphIte-ET)

In what follows, we analyze the impact of early termination (ET) by testing the GraphIte-ET implementations of community detection and PageRank.

GraphIte-ET for Community Detection rests on the main idea of terminating vertices as soon as their community labels stop changing (III.B.1). This heuristic tries to improve performance but, in the process may degrade output quality. Fig. 4 (b) shows the effect of early termination on the number of active vertices (Y-axis on the left) and on the output quality (shown as modularity by the Y-axis on the right) of community detection with GH. It should be noted that the results are shown on the sparsified inputs. We observed similar trend for other datasets as well. Fig. 4 (b) shows that as the iterations progress, the number of active vertices decreases while increasing the modularity. Table 2 shows the modularity comparison between GraphIte and GraphIte-ET for six datasets. We can see that difference in the modularity values between GraphIte and GraphIte-ET varies from 0.01% to 3.4%. Section IV.E discusses the effect of early termination on inter-PE communication volume and overall performance gain.

GraphIte-ET for PageRank rests on the main idea of terminating vertices as soon as their individual PageRank values stop changing significantly in consecutive iterations, defined by the threshold α (III.B.2). This heuristic tries to improve performance but, in the process may degrade output quality. We determine the number of active vertices and loss in precision per iteration by varying the value of α. Fig. 5 (a) shows the effect of early termination on the number of active vertices within each iteration of PageRank with GH for three threshold values of α. Here, $\alpha = 10^{-9}$ would represent a conservative threshold setting while $\alpha = 10^{-3}$ would represent an aggressive threshold setting. The results for GH are shown as examples. Fig. 5 (a) shows that a larger value of α (i.e., 10^{-3}) results in a drastic reduction in the number of active vertices but with a larger precision loss (as can be expected). On the other hand, smaller value for α (i.e., 10^{-9}) does not achieve any meaningful reduction in the number of active vertices. Fig. 5(b) illustrates the loss in precision per iteration of PageRank with GH for the three values of α mentioned above. We can see from Fig. 5(b) that as the iterations progress, we have maximum loss in precision when the value of α as 10^{-3}. In contrast, choosing 10^{-9} as the value of α achieves minimum precision loss. Moreover, careful observation of Fig. 5(b) reveals that the difference in precision loss between 10^{-6} and 10^{-9} is negligible. Hence, it is clear from Figs. 5 (a) and (b) that larger value of α achieves higher reduction in number of active vertices towards the goal of reducing computation, while a smaller value of α achieves better precision. The setting with the best tradeoff between performance and precision appears with $\alpha = 10^{-6}$ for the inputs tested. We use this setting for the full system performance on PageRank.

Table 2: Modularity comparison between GraphIte and GraphIte-ET.

Input graph (label)	GraphIte	GraphIte-ET
musae_Github (GH)	0.3593	0.3757
gemsec-Deezer (DZ)	0.8418	0.8417
road_luxembourg-osm (RM)	0.7969	0.7957
com-Orkut (OR)	0.6602	0.6835
socfb-A-anon (FB)	0.5246	0.5084
soc-LiveJournal1 (LJ)	0.7552	0.7566

E. Overall Performance Evaluation

Due to the early termination on GraphIte-ET, with progressing iterations the number of active vertices decreases. As a result, inactive vertices also stop generating the inter-PE traffic. Therefore, we study the total volume of inter-PE traffic for GraphIte and GraphIte-ET. Considering the total inter-PE traffic, GraphIte-ET reduces 48% to 66% traffic volume compared to GraphIte.

In Figs. 6 and 7, we compare the speed up and energy reduction for community detection and PageRank, respectively, on GraphIte and GraphIte-ET with respect to the Spara (baseline). In these figures, we show the range of speed up and the normalized energy for different datasets considered in this work. The full-system execution time includes the computation time, inter-PE communication time and the data transfer time from the host. Since the preprocessing step including vertex reordering using Spara followed by sparsification, is carried out in the CPU host, we exclude this preprocessing time – so that the focus of our performance analysis stays on the executions that happen on the ReRAM architecture. Figs. 6 (a) and 7 (a) show the speed up of GraphIte and GraphIte-ET with respect to Spara for community detection and PageRank, respectively. We can see from Fig. 6 (a) that GraphIte achieves 1.17x to 3.01x

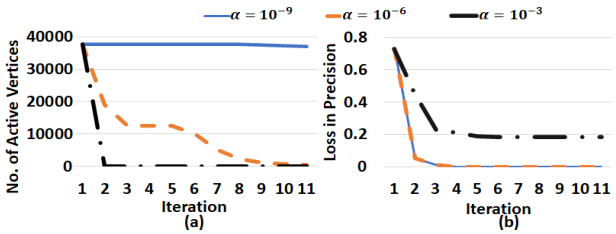

Fig. 5: PageRank: (a) Effect of early termination on the number of active vertices and (b) the loss in precision within each iteration of the algorithm for three threshold values of α with GH after sparsified inputs.

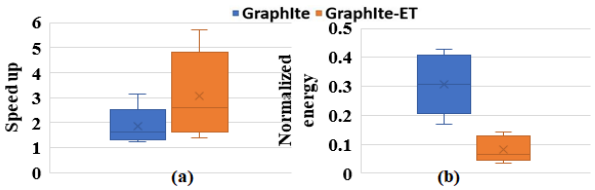

Fig. 6: (a) Speed up, (b) normalized energy of GraphIte, GraphIte-ET for community detection w.r.t Spara.

performance improvement with respect to Spara depending on the datasets considered in this work. We observed that the performance gains realized by GraphIte is higher for the larger datasets (where it matters more) – e.g., GraphIte achieves peak speedups (3.01x) for the largest input tested (LJ: 4.8M vertices and 68.9M edges). Moreover, there are more savings in total execution time achieved by GraphIte on social media datasets (1.33x to 3.01x savings) than with road network i.e., RM, CA and PA (1.17x to 1.2x) compared to Spara. The speed up is least for road network as it has a uniform degree distribution. Hence, the variation of sparsity among active blocks for road network is comparatively less than that of the social media datasets. Next, after incorporating early termination-based approximation on top of GraphIte, we can see from Fig. 6 (a) that GraphIte-ET achieves 1.3x to 5.4x speed up compared to Spara reordering (i.e., with no sparsification or early termination). Moreover, GraphIte-ET achieves 13% to 47.9% reduction in execution time compared to GraphIte depending on the considered datasets. It is clear from Fig. 6 (a) that the improvement is input dependent.

Fig. 6 (b) and Fig. 7 (b) illustrate the normalized full-system energy consumption using community detection and PageRank for GraphIte and GraphIte-ET compared to Spara. Figs. 6 (b) and 7 (b) show that GraphIte consumes 33.2% to 76.2% less energy compared to Spara. We can also see that, incorporating early termination-based approximation, GraphIte-ET achieves 18.5% to 49.3% compared to GraphIte depending on different datasets considered in this work. Moreover, GraphIte-ET outperforms Spara by consuming 45% to 87.9% less energy. As mentioned above, GraphIte and GraphIte-ET are more efficient in reducing the number of active blocks and on-chip data movement for social media datasets compared to road network, the reduction of energy consumption is least for RM.

It should also be noted that due to high energy efficiency of ReRAM-based PEs, the peak temperature of the 3D manycore system remains below 85°C for all the configurations tested. Hence, temperature hotspots are not of any concern in GraphIte and GraphIte-ET architectures.

V. Conclusion

In this paper, we demonstrated the benefits of using approximate computing for accelerating the computation as well as reducing the storage requirements of graph computations on ReRAM-based architectures. Our GraphIte implementations achieve 61% to 89% reduction of active blocks for negligible precision loss. This reduction also results in reducing the overall computation and inter-PE communication on GraphIte. Hence, it achieves 19.8% to 68.9% reduction in execution time and 33.2% to 76.2% less energy consumption compared to the state-of-the-art ReRAM-based architecture Spara (baseline). GraphIte-ET also reduces 48% to 66% of total traffic compared to

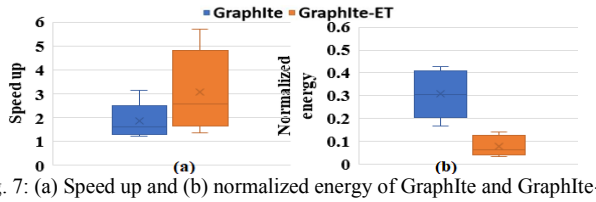

Fig. 7: (a) Speed up and (b) normalized energy of GraphIte and GraphIte-ET for PageRank with respect to Spara.

GraphIte. For full system performance evaluation, GraphIte-ET is 13% to 47.9% faster and 18.5% to 49.3% energy efficient compared to GraphIte.

Acknowledgment

This work was supported by the US National Science Foundation (NSF) grant CCF-1815467.

References

[1] S. Mittal, "A survey of techniques for approximate computing." *ACM Computing Surveys* 48, no. 4 (2016): 1-33.
[2] H. Lu, M. Halappanavar, D. Chavarria-Miranda, A. H. Gebremedhin, A. Panyala, and A. Kalyanaraman, "Algorithms for balanced graph colorings with applications in parallel computing," *IEEE Transactions on Parallel and Distributed Systems* 28, no. 5 (2016): 1240-1256.
[3] A. Panyala, O. Subasi, M. Halappanavar, A. Kalyanaraman, D. Chavarria-Miranda and S. Krishnamoorthy, "Approximate Computing Techniques for Iterative Graph Algorithms," IEEE 24th International Conference on High Performance Computing, 2017, pp. 23-32.
[4] L. Page et al., "The pagerank citation ranking: Bringing order to the web," Stanford University, Technical Report, 1998.
[5] V. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," Journal of Statistical Mechanics: Theory and Experiment, p. P10008., 2008.
[6] L. Song, Y. Zhuo, X. Qian, H. Li and Y. Chen, "GraphR: Accelerating Graph Processing Using ReRAM," IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018, pp. 531-543.
[7] G. Dai, T. Huang, Y. Wang, H. Yang, and J. Wawrzynek, "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. Association for Computing Machinery, New York, NY, USA, 2019, 120–126.
[8] L. Zheng et al., "Spara: An Energy-Efficient ReRAM-Based Accelerator for Sparse Graph Analytics Applications," 2020 IEEE International Parallel and Distributed Processing Symposium, 2020, pp. 696-707.
[9] B. K. Joardar et al., "Learning to Train CNNs on Faulty ReRAM-based Manycore Accelerators," ACM Trans. Embed. Comput. Syst. 20, 5s, Article 55, 2021, 23 pages.
[10] B. Feinberg, S. Wang, and E. Ipek, "Making memristive neural network accelerators reliable," IEEE International Symposium on High Performance Computer Architecture, Vienna, Austria, 2018, pp. 52–65.
[11] C. Lee, H. Lin, C. Lien, Y. Chih, and J. Chang, "A 1.4Mb 40-nm embedded ReRAM macro with 0.07um2 bit cell, 2.7mA/100MHz low-power read and hybrid write verify for high endurance application," IEEE Asian Solid-State Circuits Conference, 2017, pp. 9–12.
[12] Xiangyu Dong et al., "Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory," In Emerging Memory Technologies. Springer, 2014, pp. 15–50.
[13] Nan Jiang et al., "A detailed and flexible cycle-accurate Network-on-Chip simulator," 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2013, pp. 86-96.
[14] A. Shafiee et al., "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," 2016 ACM/IEEE 43rd.
[15] J. Leskovec, K. J. Lang, and M. Mahoney, "Empirical comparison of algorithms for network community detection," In Proceedings of the 19th international conference on World wide web, 2010, pp. 631-640.
[16] S. Ghosh et al., "Distributed Louvain Algorithm for Graph Community Detection," 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018.
[17] https://github.com/luhowardmark/GrappoloTK
[18] R. Bellman, "On a routing problem," Quart. Appl. Math., vol. 16, no. 1, pp. 87–90, 1958.
[19] M. M. Ozdal et al, "Energy Efficient Architecture for Graph Analytics Accelerators," Proc. International Symposium on Computer Architecture, 2016, pp. 166–177.