GraphR: Accelerating Graph Processing Using ReRAM

Linghao Song[*], Youwei Zhuo[#], Xuehai Qian[#], Hai Li[*] and Yiran Chen[*]
[*]Duke University, [#]University of Southern California
[*]{linghao.song, hai.li, yiran.chen}@duke.edu, [#]{youweizh, xuehai.qian}@usc.edu

ABSTRACT

Graph processing recently received intensive interests in light of a wide range of needs to understand relationships. It is well-known for the *poor locality* and *high memory bandwidth requirement*. In conventional architectures, they incur a significant amount of data movements and energy consumption which motivates several hardware graph processing accelerators. The current graph processing accelerators rely on memory access optimizations or placing computation logics close to memory. Distinct from all existing approaches, we leverage an emerging memory technology to accelerate graph processing with analog computation.

This paper presents GRAPHR, the first ReRAM-based graph processing accelerator. GRAPHR follows the principle of near-data processing and explores the opportunity of performing massive parallel analog operations with low hardware and energy cost. The analog computation is suitable for graph processing because: *1)* The algorithms are iterative and could inherently tolerate the imprecision; *2)* Both probability calculation (e.g., PageRank and Collaborative Filtering) and typical graph algorithms involving integers (e.g., BFS/SSSP) are resilient to errors. The key insight of GRAPHR is that if a vertex program of a graph algorithm can be expressed in sparse matrix vector multiplication (SpMV), it can be efficiently performed by ReRAM crossbar. We show that this assumption is generally true for a large set of graph algorithms.

GRAPHR is a novel accelerator architecture consisting of two components: *memory ReRAM* and *graph engine (GE)*. The core graph computations are performed in sparse matrix format in GEs (ReRAM crossbars). The vector/matrix-based graph computation is not new, but ReRAM offers the unique opportunity to realize the massive parallelism with unprecedented energy efficiency and low hardware cost. With small subgraphs processed by GEs, the gain of performing parallel operations overshadows the wastes due to sparsity. The experiment results show that GRAPHR achieves a $16.01\times$ (up to $132.67\times$) speedup and a $33.82\times$ energy saving on geometric mean compared to a CPU baseline system. Compared to GPU, GRAPHR achieves $1.69\times$ to $2.19\times$ speedup and consumes $4.77\times$ to $8.91\times$ less energy. GRAPHR gains a speedup of $1.16\times$ to $4.12\times$, and is $3.67\times$ to $10.96\times$ more energy efficiency compared to PIM-based architecture.

1. INTRODUCTION

With the explosion of data collected from massive sources, graph processing received intensive interests due to the increasing needs to understand relationships. It has been applied in many important domains including cyber security [59], social media [3], PageRank citation ranking [47], natural language processing [9, 16, 40], system biology [12, 14], recommendation systems [33, 54, 62] and machine learning [20, 42, 61].

There are several ways to perform graph processing. The distributed systems [10, 22, 30, 36, 57, 67] leverage the ample computing resources to process large graphs. However, they inherently suffer from synchronization and fault tolerance overhead [27, 50, 69] and load imbalance [63]. Alternatively, disk-based single-machine graph processing systems [28, 52, 58, 60, 70] (a.k.a. out-of-core systems) can largely eliminate all the challenges of distributed frameworks. The key principle of such systems is to keep only a small portion of active graph data in memory and spill the remainder to disks. The third approach is the in-memory graph processing. The potential of in-memory data processing has been exemplified in a number of successful projects, including RAMCloud [45], Pregel [39], GraphLab [37], Oracle TimesTen [29], and SAP HANA [19].

It is well-known for the *poor locality* because of the random accesses in traversing the neighborhood vertices, and *high memory bandwidth requirement*, because the computations on data accesses from memory are typically simple. In addition, graph operations lead to memory bandwidth waste because they only use a small portion of a cache block. In conventional architecture, graph processing incurs significant amount of data movements and energy consumption. To overcome these challenges, several hardware accelerators [4, 23, 46] are proposed to execute graph processing more efficiently with specialized architecture. In the following, we retrospect the graph computation and the current solutions to motivate our approach.

A graph can be naturally represented as an adjacency matrix and most graph algorithms can be implemented by some form of matrix-vector multiplications. However, due to the sparsity of graph, graph data are not stored in compressed sparse matrix representations, instead of matrix form. Graph processing based on sparse data representation involves: *1)* bringing data for computation from memory based on compressed representation; *2)* performing the computations on the loaded data. Due to the sparsity, the data accesses in *1)* may be *random and irregular*. In essence, *2)* performs *simple* computations that are part of the matrix-vector multiplications but only on non-zero operands. As a result, each computing core experiences alternative long random memory access latency and short computations. This leads to the well-known challenges in graph processing and other issues such as memory bandwidth waste [23].

The current graph processing accelerators mainly optimize the memory accesses. Specifically, Graphicionado [23] reduces memory access latency and improves throughput by replacing random accesses to conventional memory hierarchy with sequential accesses to scratchpad memory op-

timization and pipelining. Ozdal *et al.* [46] improves the performance and energy efficiency by latency tolerance and hardware supports for dependence tracking and consistency. TESSERACT [4] applies the principle of near-data processing by placing compute logics (e.g., in-order cores) close to memory to claim the high internal bandwidth of Hybrid Memory Cube (HMC) [48]. However, all architectures do little change on compute unit, — the simple computations are performed one at a time by instructions or specialized units.

To perform matrix-vector multiplications, two approaches exist that reflect two ends of the spectrum: *1)* the dense-matrix-based methods incur regular memory accesses and perform computations with every element in matrix/vector; *2)* the sparse-matrix-based methods incur random memory accesses but only perform computations on non-zero operands. In this paper, we adopt an approach that can be considered as the *mid-point* between these two ends that could potentially achieve better performance and energy efficiency. Specifically, we propose to perform sparse matrix-vector multiplications on data blocks of compressed representation. The benefit is two-fold. First, the computation and data movement ratio is increased. It means that the cost of bringing data could be naturally hidden by the larger amount of computations on a block of data. Second, inside this data block, computations could be performed in parallel. The downside is that certain hardware and energy will be wasted in performing useless multiplications with zero.

This approach could in principle be applied to the current GPUs or accelerators, but with the same amount of compute resources (e.g., SM in GPUs), it is unclear whether the gain would outweigh the inefficiency caused by the sparsity. Clearly, a key factor is the cost of compute logic, — with a low-cost mechanism to implement matrix-vector multiplications, the proposed approach is likely to be beneficial.

In this paper, we demonstrate that the non-volatile memory, metal-oxide resistive random access memory (ReRAM) [65] could serve as the essential hardware building block to perform matrix-vector multiplications in graph processing. Recent works [13,55,56] demonstrate the promising applications of efficient in-situ matrix-vector multiplication of ReRAM on neural network acceleration. The analog computation is suitable for graph processing because: *1)* The iterative algorithms could *tolerate the imprecise values by nature*; *2)* Both probability calculation (e.g., PageRank and Collaborative Filtering) and typical graph algorithms involving integers (e.g., BFS/SSSP) are *resilient to errors*. Due to the low-cost and energy efficiency, matrix-based computation in ReRAM would not incur significant hardware waste due to sparsity. Such waste is only incurred inside the ReRAM crossbar with moderate size (e.g., 8×8). Moreover, the sparsity is not completely lost, — if a small subgraph contains all zeros, it does not need to be processed. As a result, the architecture will mostly enjoy the benefits of more parallelism in computation and higher ratio between computation and data movements. Performing computation in ReRAM also enables near data processing with reduced data movements: the data do not need to go through the memory hierarchy like in the conventional architecture or some accelerators.

Applying ReRAM in graph processing poses a few challenges: *1) Data representation*. Graph is stored in compressed format, not in adjacency matrix, to perform in-memory computation, data needs to be converted to matrix format. *2) Large graph*. The real-world large graphs may not fit in memory. *3) Execution model*. The order of subgraph processing needs to be carefully determined because it affects the hardware cost and correctness. *4) Algorithm mapping*. It is important to map various graph algorithms to ReRAM with good parallelism.

This paper proposes, GRAPHR, a novel ReRAM-based accelerator for graph processing. It consists of two key components: memory ReRAM and graph engine (GE), which are both based on ReRAM but with different functionality. The memory ReRAM *stores* the graph data in *compressed sparse representation*. GEs (ReRAM crossbars) perform the efficient *matrix-vector multiplications* on the *sparse matrix representation*. We propose a novel *streaming-apply* model and the corresponding preprocessing algorithm to ensure correct processing order. We propose two algorithm mapping patterns, parallel MAC and parallel add-op, to achieve good parallelism for different type of algorithms. GRAPHR can be used as a drop-in accelerator for out-of-core graph processing systems.

In the evaluation, we compare GRAPHR with a software framework [70] on a high-end CPU-based platform ,GPU and PIM [4] implementations. The experiment results show that GRAPHR achieves a $16.01\times$ (up to $132.67\times$) speedup and a $33.82\times$ energy saving on geometric mean compared to the CPU baseline. Compared to GPU, GRAPHR achieves $1.69\times$ to $2.19\times$ speedup and consumes $4.77\times$ to $8.91\times$ less energy. GRAPHR gains a speedup of $1.16\times$ to $4.12\times$, and is $3.67\times$ to $10.96\times$ more energy efficiency compared to PIM-based architecture.

This paper is organized as follows. Section 2 introduces the background of graph processing, ReRAM and current hardware graph processing accelerators. Section 3 describes GRAPHR architecture. Section 4 presents processing patterns for various graph algorithms. Section 5 presents the evaluation methodology and experiment results. Section 6 concludes the paper.

2. BACKGROUND AND MOTIVATION

2.1 Graph Processing

(a) Vertex and edge accesses in graph processing   (b) Access pattern in vertex-centric program
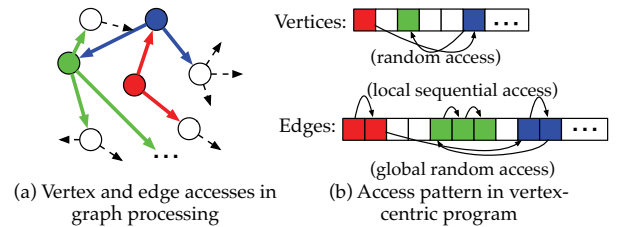
Figure 1: Graph Processing in Vertex-Centric Program

Graph algorithms traverse vertices and edges to discover interesting graph properties based on relationship. A graph could be naturally considered as an adjacency matrix, where the rows correspond to the vertices and the matrix elements represent the edges. Most graph algorithms can be mapped to matrix operations. However, in reality, the graph is *sparse*, which means that there would be many zeros in the adjacency
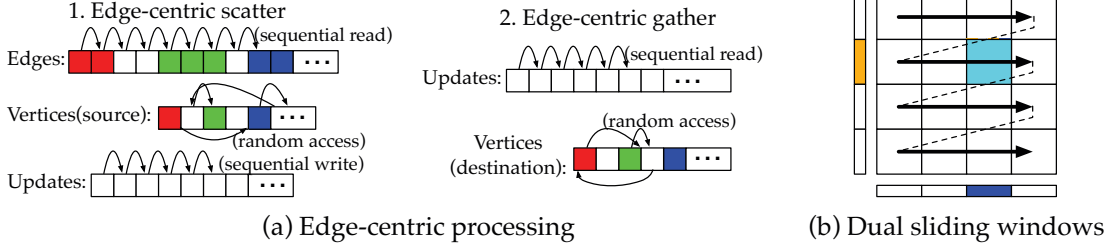
Figure 2: (a) Edge-Centric Processing and (b) Dual Sliding Windows

matrix. This property incurs the waste of both storage and compute resources. Therefore, the current graph processing systems use the format that is suitable for sparse graph data. Based on such data structures, the graph processing can be essentially considered as implementing the matrix operations on the sparse matrix representation. In this case, individual (and simple) operations in the whole matrix computation are performed by the compute units (e.g., a core in CPU or an SM in GPU) after data is fetched. In the following, we elaborate the challenge of random accesses in various graph processing approaches. More details on sparse graph data representation will be discussed in Section 2.4.

To provide an easy programming interface, the vertex-centric program featuring "Think Like a Vertex (TLAV)" [39] was proposed as a natural and intuitive way for human brain to think of the graph problems. Figure 1 (a) shows an example, an algorithm could first access and process the red vertex in the center with all its neighbors through the red edges. Then it can move to one of the neighbors, the blue vertex on the top, accessing the vertex and the neighbors through the blue edges. After that, the algorithm can access another vertex (the green one on the left), which is one of the neighbors of the blue vertex.

In graph processing, the vertex accesses lead to random accesses because the neighbor vertices are not stored continuously. For edges, they incur *local sequential access* because the edges related to a vertex are stored continuously but *global random accesses* because algorithm needs to access the edges of different vertices. The concepts are shown in Figure 1 (b). The random accesses lead to long memory latency and, more importantly, the bandwidth waste, because only a small portion of data are accessed in a memory block. In a conventional hierarchical memory system, this leads to the significant data movements.

Clearly, reducing random accesses is critical to improve the performance of graph processing, this is particularly crucial for the disk-based single machine graph processing systems (a.k.a out-of-core systems [28, 52, 60, 70]), because the random disk I/O operations are much more detrimental to the performance. In this context, the memory is considered small and fast (therefore can afford random accesses) while disk is considered large and slow (therefore should be only accessed sequentially). The edge-centric model in X-Stream [52] is a notable solution for reducing random accesses. Specifically, the edges of a graph partition are stored and accessed sequentially in disk and the related vertices are stored and accessed randomly in memory. Such setting is feasible because typically the vertex data are much smaller than edge data. During process, X-Stream generates *Updates* in *scatter* phase, which

incurs sequential writes, and then, applies these Updates to vertices in *gather* phase, which incurs sequential reads. The concepts are shown in Figure 2 (a).

A notable drawback of X-stream is that the number of updates may be as large as that of edges, GridGraph [70] proposed optimizations to further reduce the storage cost and data movements due to updates. The solution is based on the dual sliding windows (shown in Figure 2 (b)), which partitions edges into blocks and vertices into chunks. On visiting the edge blocks, the source vertex chunk (orange) is accessed and updates are directly applied to the destination vertex chunk (blue). This requires no temporary update storage as in X-Stream. Edge blocks can be accessed in source oriented order (shown in Figure 2 (b)) or destination oriented order. Note that the dual sliding window mechanism is based on edge-centric model.

2.2 ReRAM Basics

The resistive random access memory (ReRAM) [65] is an emerging non-volatile memory with appealing properties of high density, fast read access and low leakage power. ReRAM has been considered as a promising candidate for future memory architecture. A primary application of ReRAM is to be used as an alternate for main memory [18, 34, 68]. Figure 3 (a) demonstrates the metal-insulator-metal (MIM) structure of an ReRAM cell. It has a top electrode, a bottom electrode and a metal-oxide layer sandwiched between electrodes. By applying an external voltage across it, an ReRAM cell can be switched between a high resistance state (HRS or OFF-state) and a low resistance state (LRS or On-state), which are used to represent the logical "0" and "1", respectively, as shown in Figure 3 (b). The endurance of ReRAM could be up to 10^{12} [24, 31], alleviating the lifetime concern faced by other non-volatile memory, such as PCM [49].
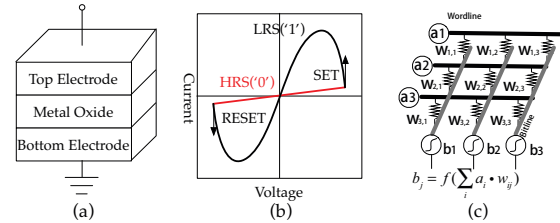

Figure 3: Basics of ReRAM

The ReRAM features the capability to perform *in-situ matrix-vector multiplication* [25, 26] as shown in Figure 3 (c), which utilizes the property of bitline current summation in ReRAM crossbars to enable computing with high performance and low energy cost. While conventional CMOS

based system showed success on neural network acceleration [6,11,38], recent works [13,35,55,56] demonstrated that ReRAM-based architectures offer significant performance and energy benefits for the computation and memory intensive neural network computing.

2.3 Graph Processing Accelerators

Due to the wide applications of graph processing and its challenges, several hardware accelerators were recently proposed. Ahn *et al.* [4] proposes TESSERACT, the first PIM-based graph processing architecture. It defines a generic communication interface to map graph processing to HMC. At any time, each core can Put a remote memory access and get interrupted to receive and execute the Put calls from other cores. Ozdal *et al.* [46] introduces an accelerator for asynchronous graph processing, which features efficient hardware scheduling and dependence tracking. To use the system, programmers have to understand its architecture and modify existing code. Graphicionado [23] is a customized graph accelerator designed for high performance and energy efficiency based on off-chip DRAM and on-chip eDRAM. It modifies the graph data structure and data path to optimize graph access patterns and designs specialized memory subsystem for higher bandwidth. These accelerators all optimize the memory accesses, reducing the latency or better tolerating the random access latency. [5] and [43] focused on the data coherence in memory which can be accessed by instructions from host CPU and the accelerator, and [43] also considered atomic operations.

2.4 Graph Representation

As discussed in Section 1, supporting the matrix-vector multiplications on small data blocks could increase the ratio between computation and data movement and reduce the pressure on memory system. With its matrix-vector multiplication capability, ReRAM could naturally perform the low-cost parallel dense operations on the sparse sub-matrices (subgraphs), enjoying the benefits without increasing hardware and energy cost.

The key insight of GRAPHR is to still store the majority of the graph data in the compressed sparse matrix representation and process the subgraphs in uncompressed sparse matrix representation. In the following, we review several commonly used sparse representations.
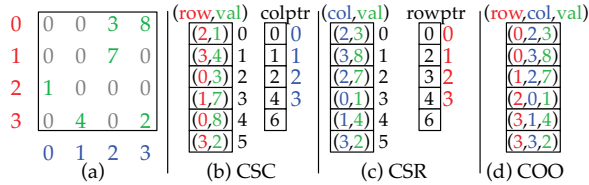

Figure 4: (a) Sparse Matrix and Its Compressed Representations in: (b) Compressed Sparse Column (CSC), (c) Compressed Sparse Row (CSR), (d) Coordinate List (COO)

The three major compressed sparse representations are compressed sparse column (CSC), compressed sparse row (CSR) and coordinate list (COO). They are illustrated in Figure 4. In the CSC representation, non-zeros are stored in column major order as (row index, value) pairs in a list,

the number of entries in the list is the number of non-zeros. Another list of column starting pointers indicate the starting index of a row in the (row,val) list. For example, in Figure 4 (a), **4** in the colptr indicates that the **4**-th entry in (row,val) list, i.e., (0,8) is the starting of column 3. The number of entries in colptr is the number of columns + 1. Compressed sparse row (CSR) is similar to CSC, with row and column alternated. For coordinate list (COO), each entry is a tuple of (row index, column index, value) of the nonzeros.
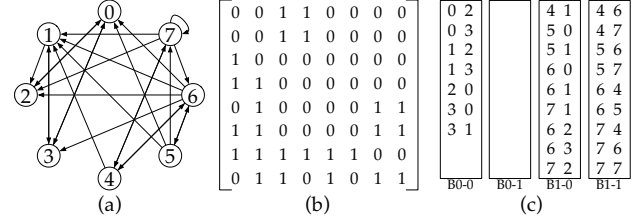

Figure 5: (a) A Directed Graph and Its Representations in (b) Adjacency Matrix and (c) Coordinate List

In GRAPHR, we assume *coordinate list (COO)* representation to *store* a graph. Given a graph in Figure 5 [1] (a), its (sparse) adjacency matrix representation and COO representation (partitioned into four 4×4 subgraphs) are shown in Figure 5(b) and (c), respectively. In this example, the coordinate list saves 61% storing space, compared with the adjacency matrix representation. For real-world graphs with high sparsity, the saving is even high: the coordinate list can only take 0.2% of space for WikiVote [32] compared to an adjacency matrix.

3. GRAPHR ARCHITECTURE

3.1 Sparse Matrix Vector Multiplication (SpMV) and Graph Processing

```
1   //Phase 1: compute edge values
2   for each edge E(V,U) from active vertex V do
3       E.value = processEdge(E.weight,V.prop)
4   end for
5
6   //Phase 2: reduce and apply
7   for each edge E(U,V) to vertex V do
8       V.prop = reduce(V.prop,E.value)
9   end for
```

Figure 6: Vertex Programming Model

Figure 6 shows the general vertex programming model for graph processing. It follows the principle of "Think Like The Vertex" [39]. In each iteration, the computation can be divided into two phases. In the first phase, each edge is *processed* with processEdge function, based on edge weight and active source vertex V's property (V.prop), it computes a value for each edge. In the second phase, each vertex *reduces* values of all incoming edges (E.value) and generates a property value, which is *applied* to the vertex as its updated property.

---

[1]As shown in Section 2.4, each entry in a coordinate list should be a three-element tuple of {Source ID, Destination ID, Edge Weight}. To simplify the example, we use an unweighted graph, so a two-element tuple of {Source ID, Destination ID} is sufficient to represent one edge.
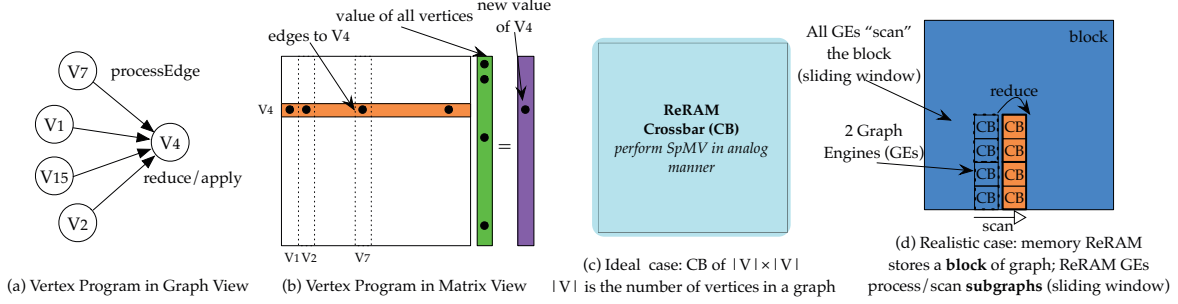
Figure 7: GRAPHR Key Insight: Supporting Graph Processing with ReRAM Crossbar

Figure 7 (a) shows vertex program in graph view. V_{15}, V_7, V_1 and V_2 each executes a processEdge function, the results are stored in corresponding edges to V_4. After all edges are processed, V_4 executes reduce function to generate the new property and update itself. In graph processing, the operation in processEdge function is multiplication, to generate the new property for each vertex, what essentially needed is the *Multiply-Accumulate (MAC)* operation. Moreover, the vertex program of each vertex is equivalent to a sparse matrix vector multiplication (SpMV) shown in Figure 7 (b). Assume A is the sparse adjacency matrix, and x is a vector containing V.prop of all vertices, the vertex program of *all vertices* can be computed *in parallel* in matrix view as $\mathbf{A^Tx}$. As shown in Figure 3 (c), ReRAM crossbar could perform matrix-vector multiplication efficiently. Therefore, as long as a vertex program can be expressed in SpMV form, it can be accelerated by our approach. We will discuss in Section 4 on the different patterns to express algorithms in SpMV.

From Figure 7 (b), we see that the size of the matrix and vector are $|V| \times |V|$ and $|V|$, respectively, where V is the number of vertices in the graph. In ideal case, if we are given a ReRAM crossbar (CB) of size $|V| \times |V|$, the vertex program of each vertex can be executed in parallel and the new value (i.e., V.prop in Figure 6) can be computed in one cycle. More importantly, the memory to store $\mathbf{A^T}$ can directly perform such in-memory computation without data movement. Unfortunately, this is unrealistic, the size of CB is quite limited (e.g., 4×4 or 8×8), to realize this idea, we need to *use Graph Engines (GEs) composed of small CBs to process subgraphs in certain manner*. This is the ultimate design goal of GRAPHR architecture. We face several key problems. First, as discussed in Section 2.4, graph is stored in compressed format, not in adjacency matrix, to perform in-memory computation, data needs to be converted to matrix format. Second, the real-world large graphs may not fit in memory. Third, the order of subgraph processing needs to be carefully determined, because this affects the hardware cost to buffer the temporary results for reduction.

To solve these problems, Figure 7 (d) shows the high level processing procedure. The GRAPHR architecture has both memory and compute module, each time, a *block* of a large graph is loaded into GRAPHR's memory module (in blue) in compressed sparse representation. If the memory module can fit the whole graph, then there is only one block, otherwise, a graph is partitioned into several blocks. Inside GRAPHR, a number of GEs (in orange), each of which is composed of a number of CBs, "scan" and process (similar to

a *sliding window*) *subgraphs* in streaming-apply model (Section 3.3). To enable in-memory computation, graph data are converted to matrix format by a controller. Such conversion is straightforward with preprocessed edge list (Section 12). The intermediate results of subgraphs are stored in buffers and simple computation logics are used to perform reduction.

GRAPHR can be used in two settings: *1) multi-node:* one can connect different GRAPHR nodes and connect them together to process large graphs. In this case, each block is processed by a GRAPHR node. Data movements happen between GRAPHR nodes. *2) out-of-core:* one can use one GRAPHR node as an in-memory graph processing accelerator to avoid using multiple threads and moving data through memory hierarchy. In this case, all blocks are processed consecutively by a single GRAPHR node. Data movements happen between disk and GRAPHR node. In this paper, we assume out-of-core setting, so we do not consider communication between nodes and its supports, we leave this as future work and extension. Next, we present the overall GRAPHR architecture and its workflow.
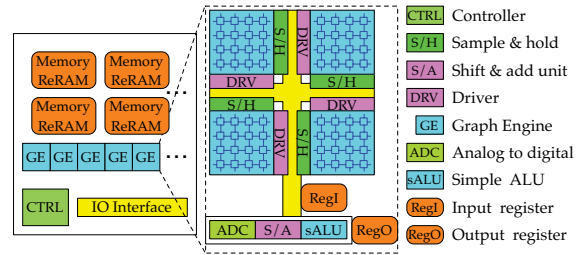
3.2 GraphR Architecture

Figure 8: GRAPHR Architecture

Figure 8 shows the architecture of one GRAPHR node. GRAPHR is a ReRAM-based memory module that performs efficient near data parallel graph processing. It contains two key components: *memory ReRAM* and *graph engine (GE)*. Memory ReRAM stores graph data in original compressed sparse representation. GEs perform efficient matrix-vector multiplications on matrix representation. A GE contains a number of ReRAM crossbars (CBs), *Drivers (DRVs)*, *Sample and Hold (S/H)* components placed in mesh, they are connected with *Analog to Digital Converter (ADC)*, *Shift and Add units (S/A)* and *simple algorithmic and logic units (sALU)*. The input and output register (*RegI/RegO*) are used to cache data flow. We discuss the detail of several components as follows.

Driver (DRV) It is used to *1)* load new edge data to ReRAM crossbars for processing; and *2)* input data into ReRAM crossbars for matrix-vector multiplication.

Sample and Hold (S/H) It is used to sample analog values and hold them before converting to a digital form.

Analog to Digital Converter (ADC) It converts analog values to digital format. Because ADCs have relatively higher area and power consumption, ADCs are not connected to every bitlines of ReRAM crossbars in a GE but shared between those bitlines. If the GE cycle is 64*ns*, we can have one ADC working at 1.0*GSps* to convert all data from eight 8-bitline crossbars within one GE.

sALU It is a simple customized algorithmic and logic unit. sALU performs operations that cannot be efficiently performed by ReRAM crossbars, such as comparison. The actual operation performed by sALU depends on algorithm and can be configured. We will show more details when we discuss algorithms in Section 4.

Data Format It is not practical for ReRAM cells to support a high resolution. Recent work [26] reported 5-bit resolution on ReRAM programing. To alleviate the pressure of diver, we conservatively assume the 4-bit ReRAM cell. To support higher computing resolution, e.g., 16 bit, the *Shift and Add (S/A)* unit is employed. A 16-bit fixed point number M can be represented as $M = [M_3, M_2, M_1, M_0]$, where each segment M_i is a 4-bit number. We can shift and add results from four 4-bit-resolution ReRAM crossbars, i.e. $D_3 \ll 12 + D_2 \ll 8 + D_1 \ll 4 + D_0$ to get a 16-bit result.

When sALU and S/A are bypassed, a graph engine could be simply considered as a memory ReRAM mat. A similar scheme of reusing ReRAM crossbars for computing and storing is employed in [13].

The I/O interface is used to load graph data and instructions into memory ReRAM and controller, respectively. In GRAPHR, *controller* is the software/hardware interface that could execute simple instructions to: *1)* coordinate graph data movements between memory ReRAM and GEs based on streaming-apply execution model; *2)* convert edges in preprocessed coordinate list (assumed in this paper but can work with other representations as well) in memory ReRAM to sparse matrix format in GEs; *3)* perform convergence check.
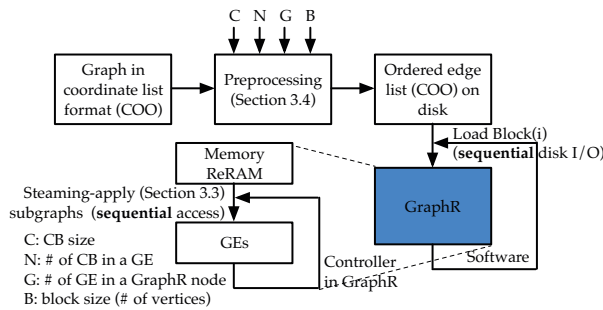

Figure 9: Workflow of GRAPHR in Out-Of-Core Setting

Figure 9 shows the workflow of GRAPHR in an out-of-core setting. GRAPHR node can be used as a drop-in in-memory graph processing accelerator. The loading of each block is performed in software by an out-of-core graph processing framework (e.g., GridGraph [70]). The integration is easy because it already contains the codes to load block from disk to

```
1  while(true)
2      load edges for next subgraph into GEs;
3      process (processEdge) in GE;
4      reduce by sALU;
5      if(check_convergence())
6          break;
7  }
```

Figure 10: Controller Operations

DRAM, we just need to redirect data to GRAPHR node. Since edge list is preprocessed in certain order, loading each block only involves sequential disk I/O. Inside GRAPHR node, the data is initial loaded into memory ReRAM, the controller manages the data movements between GEs in streaming-apply manner. Edge list preprocessing for GRAPHR needs to be carefully designed and it is based on the architectural parameters. Figure 10 shows the operations performed by controller.

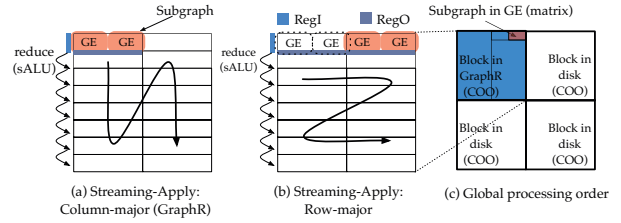### 3.3 Streaming-Apply Execution Model

Figure 11: Streaming-Apply Execution Model

In GRAPHR, all GEs process a subgraph, the order of processing is important, since it affects hardware resource requirement. We propose *streaming-apply* execution model shown in Figure 11. The key insight is that subgraphs are processed in a streaming manner and the reductions are performed on the fly by sALU. There are two variants of this model: column-major and row-major. During execution, *RegI* and *RegO* are used to store source vertices and updated destination vertices. In column-major order in Figure 11 (a), subgraphs with same destination vertices are processed together. The required *RegO* size is the same as the number of destination vertices in a subgraph. In row-major order in Figure 11 (b), subgraphs with same source vertices are process together. The required *RegO* size is the total number of destination vertices of all subgraphs with the same source vertices. It is easy to see that row-major order requires larger *RegO*. On the other side, row-major order incurs less read of *RegI*, — only one read is needed for all subgraphs with same source vertices. In GRAPHR, we use column-major order since it requires less registers, and in ReRAM, the write cost is higher than read cost.

Figure 11 (c) shows the global processing order in a complete out-of-core setting. We see that the whole graph is partitioned into 4 blocks, three of them are stored in disk in COO representation, one is stored in GraphR's memory ReRAM also in COO representation. Only the subgraph being processed by GEs is in sparse matrix format. Due to the sparsity of graph data, if the subgraph is empty, then GEs can move down to the next subgraph. Therefore, the sparsity only incurs waste inside the subgraph (e.g., when one GE has an empty matrix but others do not).

3.4 Graph Preprocessing

To support streaming-apply execution model and conversion from coordination list representation to sparse matrix format, edge list needs to be preprocessed so that edges of consecutive subgraphs are stored together. It also ensures sequential disk and memory access on block/subgraph loading. In the following, we explain the preprocessing procedure in detail.
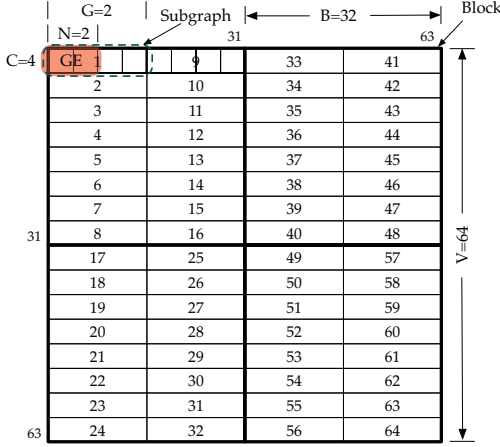

Figure 12: Preprocessing Edge List

Given some architectural parameters, the preprocessing is implemented in software and performed only once. Figure 12 illustrates these parameters and the subgraph order we should produce. This order is determined by streaming-apply model. C is the size of CB; N is the number of CBs in a GE; G is the number of GEs in a GRAPHR node; and B is the number of vertices in a block (i.e., block size). Also, V is the number of vertices in the graph. Among the parameters, C, N and G specify the architecture setting of a GRAPHR node, B is determined by the size of memory ReRAM in the node. In the example, the graph has 64 vertices ($V = 64$) and each block has 32 vertices ($B = 32$), so the graph is partitioned into 4 blocks, each one will be loaded from disk to GRAPHR node. Further, $C = 4, N = 2, G = 2$, so the subgraph size is $C \times (C \times N \times G) = 4 \times (4 \times 2 \times 2) = 4 \times 16$. Therefore, each block is partitioned into 16 subgraphs, the number of each is the global subgraph order. Our goal is to produce an ordered edge list according this order so that edges could be loaded sequentially.

Before presenting the algorithm, we assume that the original edge list is first ordered based on source vertex and then for all edges with the same source, they are ordered by destination. In another word, edges are stored in row-major order in matrix view. We also assume that in the ordered edge list, edges in a subgraph is stored in column-major order. Considering the problem in matrix view, each edge (i, j), we first compute a global order ID ($I_{(i,j)}$), so we have a 3-tuple: $(i, j, I_{(i,j)})$. This global order ID takes all zeros into account, for example, if there are k zeros between two edges in the global order, the different of global order ID of them is still k. Then, all 3-tuples could be sorted by $I_{(i,j)}$. The ordered edge list is obtained if we output them according to this order. The space and time complexity of this procedure are $O(V)$ and $O(V \log V)$, respectively, with common method. The key problem here is to compute $I_{(i,j)}$ for each (i, j). Without confusion, we simply denote $I_{(i,j)}$ as I. We show that I can be computed in a hierarchical manner.

Let B_I denote the global block order of the block that contains (i, j). We assume that blocks are also processed in column-major order: $B_{(0,0)} \rightarrow B_{(1,0)} \rightarrow B_{(0,1)} \rightarrow B_{(1,1)}$. The coordinates of a block B are:

$$B_i = \left\lfloor \frac{i}{B} \right\rfloor, B_j = \left\lfloor \frac{j}{B} \right\rfloor \tag{1}$$

Based on column-major order, B_I is:

$$I_B = B_j + (V/B) \times B_j \tag{2}$$

We assume that B can divide V, and similarly, C can divide B, $C \times N \times B$ can divide B. Otherwise, we can simply pad zeros to satisfy the conditions, it will not affect the results since these zeros do not correspond to actual edges. The block corresponding to B_I start with the following global order ID:

$$start\_global\_ID(B_I) = B_I \times B^2/(C^2 \times N \times G) + 1 \tag{3}$$

Next, we compute SI, (i, j)'s global subgraph ID. The coordinates of the edge from the start of a block B_I (B_i, B_j) are:

$$i' = i - B_i \times B, j' = j - B_j \times B \tag{4}$$

The relative coordinates of the subgraph from the start of correspond block are:

$$SI_{i'} = \left\lfloor \frac{i'}{C} \right\rfloor, SI_{j'} = \left\lfloor \frac{j'}{C \times N \times G} \right\rfloor \tag{5}$$

Then, we can compute SI:

$$\begin{aligned} SI &= (S_{i'} + S_{j'} \times B/C) + start\_global\_ID(B_I) \\ &= (S_{i'} + S_{j'} \times B/C) + B_I \times B^2/(C^2 \times N \times G) + 1 \end{aligned} \tag{6}$$

Finally, we compute $SubI$, the relative order of (i, j) from its corresponding subgraph (SI). The coordinates are:

$$\begin{aligned} SubI_i &= i - (B \times B_i) - (S_i \times C), \\ SubI_j &= j - (B \times B_j) - (S_j \times C) \end{aligned} \tag{7}$$

Since edges in a subgraph are assumed to be stored in column-major order, $SubI$ is:

$$SubI = SubI_i + (SubI_j - 1) \times C \tag{8}$$

With SI and $SubI$ computed, we get I:

$$I = (SI - 1) \times (C^2 \times N \times G) + SubI \tag{9}$$

3.5 Discussion

```
1   //Phase 1: compute edge values
2   for each edge E(V,U) from active vertex V do
3       E.value = r * V.prop / V.outdegree
4   end for
5
6   //Phase 2: reduce and apply
7   for each edge E(U,V) to vertex V do
8       V.prop = sum(E.value) + (1-r) / Num_Vertex
9   end for
```

Figure 13: PageRank in Vertex Program

Table 1 compares different architectures for graph processing. GRAPHR improves over previous architectures due to

	CPU	GPU	Tesseract [4]	GAA [46]	Graphicionado [23]	**GRAPHR**
Process Edge	Instruction	Instruction	Instruction	Specialized AU	Specialized unit	ReRAM crossbar
Reduce	Instruction	Instruction	Instruction and inter-cube communication	Specialized APU/SCU	Specialized unit	ReRAM crossbar or sALU
Processing Model	Sync/Async	Sync	Sync	Async	Sync	Sync
Data Movement	Disk to memory (out-of-core); Memory hierarchy	Disk to memory; CPU/GPU memory; GPU mem. hierarchy	Between cubes (in-memory only)	Between memory and accelerator (in-memory only)	Between modules in memory pipeline; memory to SPM; SPM to/from processing units.	Disk to memory (out-of-core) or between GRAPHR nodes (multi-node); Between memory ReRAM and GEs (inside GRAPHR)
Memory Access	Random: vertex access and start of edge list of a vertex; Sequential: edge list.				Reduced random with SPM. Pipelined memory access	Sequential edge list.
Generality	All algorithms		Vertex program			Vertex program in SpMV

Table 1: Comparison of Different Architectures for Graph Processing

```
1   //Phase 1: compute edge values
2   for each edge E(V,U) from active vertex V do
3       E.value = E.weight + V.prop
4       activate(U);
5   end forSomes
6
7   //Phase 2: reduce and apply
8   for each edge E(U,V) to vertex V do
9       V.prop = min(V.prop,E.value)
10  end for
```

Figure 14: SSSP in Vertex Program

two unique features. First, the computation is performed in analog manner, others use either instructions or specialized digital processing units. This provides the excellent energy efficiency. Second, all disk and memory accesses in GRAPHR are sequential, this is due to preprocessing and less flexibility in scheduling vertices. It is a good trade-off because it is highly energy efficient to perform parallel operations in ReRAM CBs. We believe that GRAPHR is the first architecture scheme using ReRAM CBs to perform graph processing, and the paper presents detailed and complete solution to integrate it as a drop-in accelerator for out-of-core graph processing systems. The architecture, streaming-apply execution model and the preprocessing algorithms are all novel contributions. Also, GRAPHR is general because it could accelerate all vertex programs that can be performed in SpMV form.

4. MAPPING ALGORITHMS IN GE

In this section, we discuss two patterns when mapping algorithms to GEs: parallel MAC and parallel add-op. We use a typical example for each category (i.e., PageRank and SSSP, respectively) to explain the insights. More examples (but not all) of supported algorithms in GRAPHR are listed in Table 2. The first two are parallel MAC pattern and the last two are parallel add-op pattern.

4.1 Parallel MAC

In an algorithm, if processEdge function performs a multiplication, which can be performed in *each CB cell*, we call it *parallel MAC* pattern. The parallelization degree is roughly $(C \times C \times N \times G)$ (see parameter in Figure 9).

PageRank [47] is an excellent example of this pattern. Figure 13 shows its vertex program. It does the following iterative computation: $\overrightarrow{PR}_{t+1} = r\mathbf{M} \cdot \overrightarrow{PR}_t + (1-r)\overrightarrow{e}$. $\overrightarrow{PR}_t$ is

the PageRank at iteration t, $\mathbf{M}$ is a probability transfer matrix, r is the probability of random surfing and $\overrightarrow{e}$ is a vector of probabilities of staying in each page.

We consider a small subgraph that could be processed by a 5×4 CB (the additional row is to perform addition). It contains at most 16 edges represented by the intersection of 4 rows and 4 columns in the sparse matrix (shown in Figure 16 (a)). Thus, the block is related to 8 vertices (i.e., $i0 \sim i3$, $j0 \sim j3$). The following are the parameters for the 4×4 block PageRank shown in Figure 16 (b1): $\mathbf{M} = [0,1/2,1,0;1/3,0,0,1/2;1/3,0,0,1/2;1/3,1/2,0,0]$, $\overrightarrow{e} = [1/4,1/4,1/4,1/4]^{\mathrm{T}}$, $r = 4/5$.

We define $\mathbf{M_0} = r\mathbf{M}$ and $\overrightarrow{e_0} = (1-r)\overrightarrow{e}$, so that $\overrightarrow{PR}_{t+1} = \mathbf{M_0}\overrightarrow{PR}_t + \overrightarrow{e_0}$. In CB in Figure 16 (b2, b3), the values are already scaled with r. Figure 16 (b3) shows the mapping of PageRank algorithm to a CB. The additional row is used to implement the addition of $\overrightarrow{e_0}$. The sALU is configured to perform add operation in the reduce function to add PageRank values (Figure 15 (a)). To check convergence, the new PageRank value is compared with it in the previous iteration, the algorithm is converged if the difference is less than a threshold.
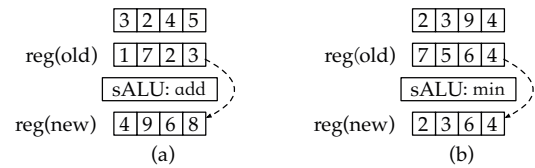

Figure 15: sALU Is Configured to Perform (a) add **in PageRank and (b)** min **in SSSP**

4.2 Parallel Add-Op

In an algorithm, if processEdge function performs an addition, which can be performed in for *each CB row* at a time, we call it *parallel add-op* pattern. The *op* specifies the operation in reduce that is implemented in sALU. The parallelization degree is roughly $(C \times N \times G)$. Single Source Shortest Path (SSSP) [15] is a typical example of this pattern. Figure 14 shows the vertex program. We see that processEdge performs an addition and reduce performs *min* operation. Therefore, sALU is configured to perform min operation shown in Figure 15 (b).

In SSSP, each vertex v is given a distance label $dist(v)$ that maintains the length of the shortest known path to vertex v
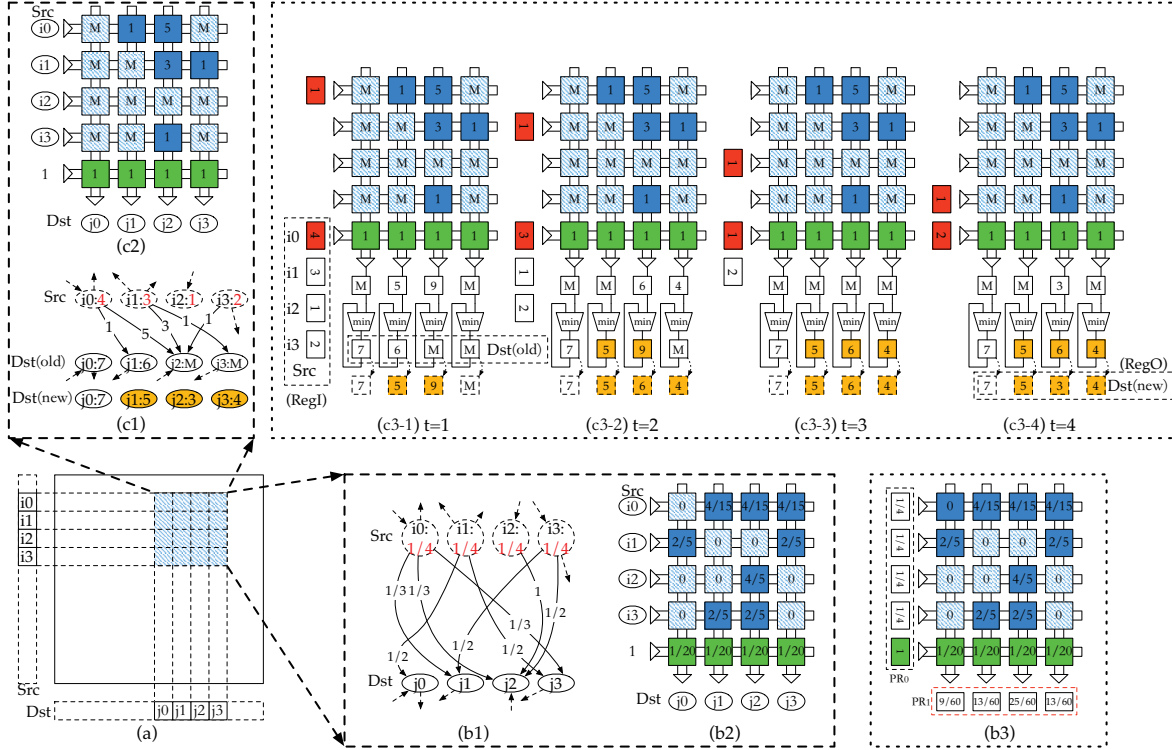
Figure 16: Graphs to Illustrate (a) PageRank and (b) SSSP

from the source. The distance label is initialized to 0 at the source and ∞ at all other nodes. Then the SSSP algorithm iteratively applies the *relaxation operator*, which is defined as follows: if (u, v) is an edge and $dist(u) + w(u, v) < dist(v)$, the value of $dist(v)$ is updated to $dist(u) + w(u, v)$. An active vertex is relaxed by applying this operator to all the edges connected to it. Each relaxation may lower the distance label of some vertex, and when no further lowering can be performed anywhere in the graph, the resulting vertex distance labels indicate the shortest distances from the source to the vertex, regardless of the order in which the relaxations were performed. Breadth-first numbering of a graph is a special case of SSSP where all edge labels are 1.

We explain the mapping of SSSP algorithm to a CB using a small 8-vertex subgraph corresponding to a 4×4 block in sparse matrix, as shown in Figure 16 (c1). It can be represented by adjacency matrix: $\mathbf{W} = [M, 1, 5, M; M, M, 3, 1; M, M, M, M; M, M, 1, M]$ where M indicates no edge connected two vertices and M is set to a reserved maximum value for a memory cell in a CB. The values are stored in the CB shown in Figure 16 (c2).

Given a vertex u and $dist(u)$, the row in the adjacency matrix $\mathbf{W}$ for u indicates $w(u, v)$. We could *perform the relaxation operator (i.e., addition) for u in parallel.* Here, SpMV is only used to select a row in CB by multiplying with an one-hot vector.

The relaxation operator of u involves *reading*: *1) dist(u)*: it is computed iteratively from the source, for the source vertex, the initial value is *zero*; *2)* The *vector* of the $dist(v)$ before the relaxation operator: it is a vector indicating the distance between source and all other vertices and is also computed iteratively from the source. In our example, for

the source vertices $(i0, i1, i2, i3)$, the initial value is $[4, 3, 1, 2]$; *3)* The *vector* of the $w(u, v)$: it is the distance from u to the destination vertices in the subgraph, and can be obtained by reading a row in adjacency matrix $\mathbf{W}$.

Figure 16 (c3) shows the process to perform SSSP in a 5 $\times$ 4 CB. The last row (green squares) is set to a fixed value 1, which is used to add $dist(u)$ (the input to the last wordline) to each $w(u, v)$ in the relaxation operator. The initial value for $dist(u)$ for the destination vertices $(j0, j1, j2, j3)$ are $[7, 6, M, M]$. The vector of $w(u, v)$ is obtained by activating the wordline associated to vertex v. In the time slot $(t = 1)$, wordline 0 (for source vertex $i0$) is activated (the red square with input value 1) and a value 4 (this is the current value in $dist(v)$ for source $i0$) is input to the last wordline (the green box line). The vector of the $dist(v)$ for the destination vertices $((j0, j1, j2, j3))$ is set as the value of the output at each bitline, which is $[7, 6, M, M]$. With this mapping, the current summation in bitline in Figure 16 (c3-1) is $[1 \times M + 4 \times 1, 1 \times 1 + 4 \times 1, 1 \times 5 + 4 \times 1, 1 \times M + 4 \times 1] = [M, 5, 9, M]$. It is the $dist(u) + w(u, v)$ computed in parallel, where u is the source vertex. Then the distance of source to each vertex v is compared with the initial $dist(v)$ $([7, 6, M, M])$ by an array of comparators, and in the final output of bitline, we get $[7, 5, 9, M]$, which is the updated $dist(v)$ vector after time slot $t = 1$. The parallel comparisons are performed by vertex-related operations in. Different algorithms may require different functions on vertices.

After time slot $t = 1$, we move to the next vertex in Figure 16 (c3-2), where we *i)* activate the second wordline; *ii)* change the input to the last wordline to 3 (the distance label for source vertex $i1$); and *iii)* set the intermediate $dist(v)$ to be the final output of bitline in time slot $t = 1$, which

Applications	Vertex Property	processEdge()	reduce()	Ative Vertex List
SpMV	Multiplication Value	E.value = V.prop / V.outdegree * E.weight	V.prop = sum(E.value)	Not Required
PageRank	Page Rank Value	E.value = r * V.prop / V.outdegree	V.prop = sum(E.value) + (1-r) / Num_Vertex	Not Required
BFS	Level	E.value = 1 + V.prop	V.prop = min(V.prop,E.value)	Required
SSSP	Path Length	E.value = E.weight + V.prop	V.prop = min(V.prop,E.value)	Required

Table 2: Property and Operations of Applications in GRAPHR

is $[7,5,9,M]$. Similar as Figure 16 (c3-1), the the current summation in bitline is $[M,M,6,4]$ and it is compared with $[7,5,9,M]$, yielding the final output of bitline for time slot $t = 2$ as $[7,5,6,4]$. We indicate the updated distance label using orange squares. The operations in time slot $t = 3$ and $t = 4$ can be performed in the similar manner.

Initially, before processing the block, the active indicator for each destination vertex is set to be FALSE. After the serial processing in CB, the active indicators for all vertices which have been updated (marked in orange in Figure 16 (c3)) are set to be TRUE. This indicates that they are active for next iteration. In our example, $j1$, $j2$, $j3$ are marked active. Referring to Figure 16, this means that the distance labels for these vertices have been updated. Because we are processing the block in CB, the active indicator for destination vertex may be accessed for multiple times, but if it is set be TRUE at least one time, this vertex is active in next iteration. Globally, after all active source vertices and corresponding edges are processed in an iteration, source vertex properties (values and active indicators) that hold the old values are updated by the properties of the same vertices in destination. The new source vertex properties are used in the next iteration. We can check if there are still active vertices to determine the convergence.

5. EVALUATION

5.1 Graph Datasets and Applications

Dataset	# Vertices	#Edges
WikiVote(WV) [32]	7.0K	103K
Slashdot(SD) [32]	82K	948K
Amazon(AZ) [32]	262K	1.2M
WebGoogle(WG) [32]	0.88M	5.1M
LiveJournal(LJ) [32]	4.8M	69M
Orkut(OK) [51]	3.0M	106M
Netflix(NF) [8]	480K users, 17.8K movies	99M

Table 3: Graph Datasets

Table 3 shows the datasets used in our evaluation. We use seven real-world graphs. For WikiVote(WV), Slashdot(SD), Amazon(AZ), WebGoogle(WG), LiveJournal(LJ), Orkut(OK) and Netflix(NF). We run pagerank(PR), breadth first search(BFS), single source shortest path (SSSP) and sparse matrix-vector multiplication (SpMV) on the first six datasets. On Netflix(NF), we run collaborative filtering(CF), and the feature length used is 32.

5.2 Experiment Setup

In our experiments, we compare GRAPHR with a CPU baseline platform, a GPU platform and PIM-based architecture [4]. PR, BFS, SSSP and SpMV running on the CPU platform are based on the software framework GridGraph [70], while collaborative filtering is based on GraphChi [28]. PR, BFS, SSSP and SpMV running on GPU platform are based on Gunrock [64], while CuMF_SGD [66] is the GPU framework for CF. We evaluate PIM-based architecture on zSim [53], a

scalable x86-64 multicore simulator. We modified zSim with HMC memory and interconnection model, heterogeneous compute units, on-chip network and other hardware features. The results are validated results with NDP [21], which also extends zSim for HMC simulation. In all experiments, graph data could fit in memory. We also exclude the disk I/O time from the execution time of the CPU/GPU-based platform.

CPU	Intel Xeon E5-2630 V3, 8 cores, 2.40 GHz $8 \times (32 + 32)$KB L1 Cache 8×256KB L2 Cache 20 MB L3 Cache
Memory	128 GB
Storage	1 TB
2 CPUs, a total number of 32 threads.	

Table 4: Specifications of the CPU Platform

Graphic Card	NVIDIA Tesla K40c
Architecture	Kepler
# CUDA Cores	2880
Base Clock	745 MHz
Graphic Memory	12 GB GDDR5
Memory Bandwidth	288 GB/s
CUDA Version	7.5

Table 5: Specifications of the GPU Platform

Specifications of the CPU and GPU platforms are shown in Table 4 and Table 5. The CPU energy consumption is estimated by Intel Product Specifications [2] while NVIDIA System Management Interface (nvidia-smi) is used to estimate energy consumption by GPU. The execution times for CPU/GPU platform are measured in the computing frameworks.

To evaluate GRAPHR, for the ReRAM part, we use NVSim [17] to estimate time and energy consumption. The HRS/LRS resistance are 25M/50K Ω, read voltage (Vr) and write voltage (Vw) are 0.7V and 2V respectively, and current of LRS/HRS are 40 uA and 2 uA respectively. The read/write latency and read/write energy cost used are 29.31ns / 50.88ns, 1.08 pJ / 3.91nJ respectively from data reported in [44]. The programming of a bipolar ReRAM cell is to change (from High to Low) or inverse. For multi-level cell, the programming is to change the resistance to a middle state between High and Low, and the middle state is determined by the programming voltage pulse length. Actually, the difference between High and Low is the *worse case*. Note that [7, 26] describe a ReRAM programming prototype, which includes: 1) writing circuitry; 2) ReRAM cell/array; and 3) conversion circuitry. They demonstrated the *possibility* of 1% accuracy for multi-level cell. However, in a real production system, only "writing circuitr" and "ReRAM cell/array" are needed, there is no need to consider the conversion, as we just need to "acquiesce" a writing precision. Therefore, this energy cost estimation for 4-bit cell programming is reasonable and more conservative than two recent ReRAM-based accelerators [13, 55].

For on-chip registers, we use CACTI 6.5 [1] at 32nm to model. For Analog/Digital converters, we use data from [41].

The system performance is modeled by code instrumentation. The ReRAM crossbar size S, number of ReRAM crossbars per graph engine C and number of graph engines is 8, 32, 64 respectively.

5.3 Performance Results

Figure 17 compares the performance of GRAPHR and CPU platform. The CPU implementation is used as the baseline and execution times of applications of GRAPHR are normalized to it. Compared to CPU platform, the geometric mean of speedups with GRAPHR architecture on all 25 executions is $16.01\times$. Among all applications on the datasets, the highest speedup achieved by GRAPHR is $132.67\times$, and it happens on SpMV on WikiVote dataset. The lowest speedup GRAPHR achieved is $2.40\times$, on SSSP using OK dataset. PageRank and SpMV are parallel MAC pattern and have higher speedup compared to CPU-based platform. For BFS and SSSP which are parallel add-op pattern, GRAPHR achieves lower speedups only due to parallel addition.
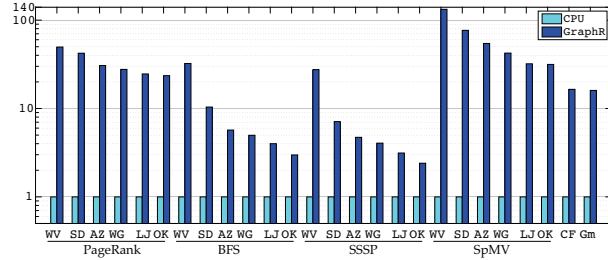

Figure 17: GRAPHR Speedup Compared to CPU
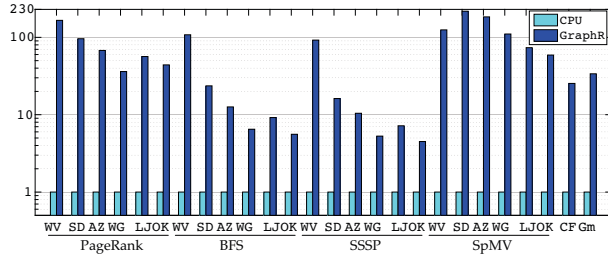
5.4 Energy Results

Figure 18: GRAPHR Energy Saving Normalized to CPU Platform

Figure 18 shows the energy savings in GRAPHR architecture over CPU platform. The geometric mean of energy savings of all applications compared to CPU is $33.82\times$. The highest energy achieved by GRAPHR is $217.88\times$, which is on sparse matrix-vector multiplication on SD dataset. The lowest energy saving achieved by GRAPHR happens on SSSP on OK dataset, which is $4.50\times$. GRAPHR gets the high energy efficiency from the non-volatile property of ReRAM and the in-situ computation capability.

5.5 Comparison to GPU Platform

GPUs take advantage of a large amount of threads (CUDA cores) for high parallelism. The GPU used in the comparison has 2880 CUDA cores, while in GRAPHR we have a comparable number ($2048 = 32 \times 64$) of crossbars. To compare with GPU, we run PageRank and SSSP on LiveJournal dataset, and collaborative filtering(CF) on Netflix dataset.
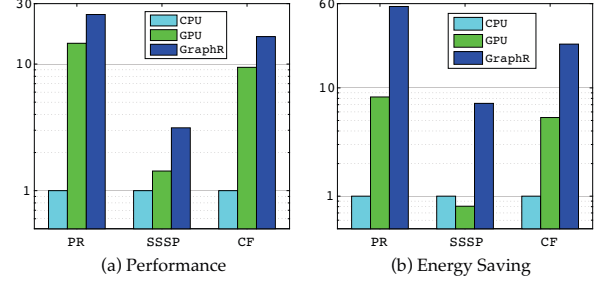

Figure 19: GRAPHR (a) Performance and (b) Energy Saving Compared to GPU Platform

The performance and energy saving normalized to CPU are shown in Figure 19 (a) and (b). Overall, the performance of GRAPHR is higher than GPU with *considering the data transfer time between CPU memory and GPU memory*, — an overhead GRAPHR does not incur. GRAPHR has performance gains ranging from $1.69\times$ to $2.19\times$ compared to GPU. More importantly, GRAPHR consumes $4.77\times$ to $8.91\times$ less energy than GPU. Figure 19 (a) shows that, GRAPHR achieves higher speedups on PageRank and CF, where MACs dominate the computation and are fully supported by GRAPHR and GPU. For SSSP, the vertex-related traversing in GRAPHR requires accessing to main memory and storage. In GPU, a cache based memory hierarchy better supports the random accessing. So the speedup of GRAPHR is lower. The reason why GRAPHR still has gain in SSSP is due to sequential access pattern. For energy saving, GRAPHR is better than GPU. Besides energy saving by in-situ computation in GEs and the in-memory processing system design, from Fig 17 in [23] we see that in conventional CMOS system, static energy consumption by eDRAM (memory) incurs the majority of energy consumption. As the technology node scales down, leakage power dominates in CMOS system. In contrast, ReRAM has almost 0 static energy leakage, so GRAPHR has higher energy saving.
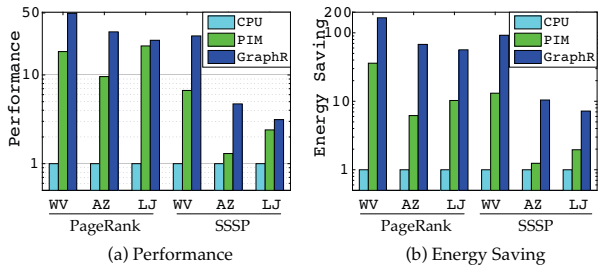
5.6 Comparison to PIM Platform

Figure 20: GRAPHR (a) Performance and (b) Energy Saving Compared to PIM Platform

We compare GRAPHRwith a PIM-based architecture (i.e., Tesseract [4]). The performance and energy saving normalized to CPU are shown in Figure 20 (a) and (b). GRAPHR gains a speedup of $1.16\times$ to $4.12\times$, and is $3.67\times$ to $10.96\times$ more energy efficiency compared to PIM-based architecture.

5.7 Sensitivity to Sparsity

We use $\#Edge/(\#Vertex)^2$ to represent the density of a dataset, and with the density decreases, the sparsity increases. Figure 21 (a) and (b) shows the performance and energy sav-

ing of GRAPHR(compared to the CPU platform) with the density of datasets. With the sparsity increases, the performance and energy saving slightly decreases. Because with the sparsity increases, the number of edge blocks to be traversed will increase, which slows down the edge accessing time and consumes more energy.
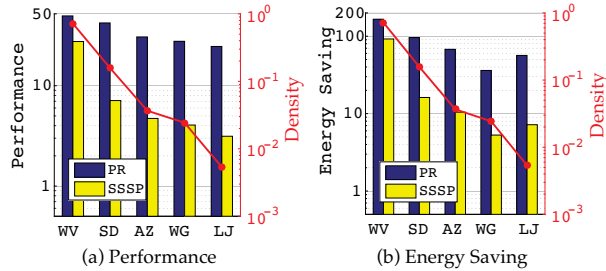

(a) Performance   (b) Energy Saving

Figure 21: GRAPHR (a) Performance and (b) Energy Saving with Dataset Density

6. CONCLUSION

This paper presents GRAPHR, the first ReRAM-based graph processing accelerator. The key insight of GRAPHR is that if a vertex program of a graph algorithm can be expressed in sparse matrix vector multiplication (SpMV), it can be efficiently performed by ReRAM crossbar. GRAPHR is a novel accelerator architecture consisting of two components: *memory ReRAM* and *graph engine (GE)*. The core graph computations are performed in sparse matrix format in GEs (ReRAM crossbars). With small subgraphs processed by GEs, the gain of performing parallel operations overshadows the wastes due to sparsity. The experiment results show that GRAPHR achieves a $16.01\times$ (up to $132.67\times$) speedup and a $33.82\times$ energy saving on geometric mean compared to a CPU baseline system. Compared to GPU, GRAPHR achieves $1.69\times$ to $2.19\times$ speedup and consumes $4.77\times$ to $8.91\times$ less energy. GRAPHR gains a speedup of $1.16\times$ to $4.12\times$, and is $3.67\times$ to $10.96\times$ more energy efficiency compared to PIM-based architecture.

REFERENCES

[1] "Cacti," http://www.hpl.hp.com/research/cacti/.

[2] "Intel xeon processor e5-2630 v3 (20m cache, 2.40 ghz)," http://ark.intel.com/products/83356/Intel-Xeon-Processor-E5-2630-v3-20M-Cache-2_40-GHz.

[3] E. Agichtein, C. Castillo, D. Donato, A. Gionis, and G. Mishne, "Finding high-quality content in social media," in *Proceedings of the 2008 international conference on web search and data mining*. ACM, 2008, pp. 183–194.

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 105–117.

[5] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd International Symposium on*. IEEE, 2015, pp. 336–348.

[6] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep neural network

computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 1–13.

[7] F. Alibart, L. Gao, B. D. Hoskins, and D. B. Strukov, "High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm," *Nanotechnology*, vol. 23, no. 7, p. 075201, 2012.

[8] J. Bennett and S. Lanning, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007, 2007, p. 35.

[9] C. Biemann, "Chinese whispers: an efficient graph clustering algorithm and its application to natural language processing problems," in *Proceedings of the first workshop on graph based methods for natural language processing*. Association for Computational Linguistics, 2006, pp. 73–80.

[10] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 1.

[11] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[12] E. J. Chesler, L. Lu, S. Shou, Y. Qu, J. Gu, J. Wang, H. C. Hsu, J. D. Mountz, N. E. Baldwin, M. A. Langston *et al.*, "Complex trait analysis of gene expression uncovers polygenic and pleiotropic networks that modulate nervous system function," *Nature genetics*, vol. 37, no. 3, pp. 233–242, 2005.

[13] P. Chi, S. Li, Z. Qi, P. Gu, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of ISCA*, vol. 43, 2016.

[14] A. Conesa, S. Götz, J. M. García-Gómez, J. Terol, M. Talón, and M. Robles, "Blast2go: a universal tool for annotation, visualization and analysis in functional genomics research," *Bioinformatics*, vol. 21, no. 18, pp. 3674–3676, 2005.

[15] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.

[16] S. H. Corston, W. B. Dolan, L. H. Vanderwende, and L. Braden-Harder, "System for processing textual inputs using natural language processing techniques," May 31 2005, uS Patent 6,901,399.

[17] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, vol. 31, no. 7, 2012.

[18] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara *et al.*, "19.7 a 16gb reram with 200mb/s write and 1gb/s read in 27nm technology," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 338–339.

[19] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: data management for modern business applications," *ACM Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012.

[20] B. J. Frey, *Graphical models for machine learning and digital communication*. MIT press, 1998.

[21] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 2015, pp. 113–124.

[22] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 17–30.

[23] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of the 49th International Symposium on Microarchitecture*. ACM, 2016.

[24] C. Hsu, I. Wang, C. Lo, M. Chiang, W. Jang, C. Lin, and T. Hou, "Self-rectifying bipolar taox/tio2 rram with superior endurance over 1012 cycles for 3d high-density storage-class memory vlsi tech," in *2013 Symposium on*, 2013, pp. T166–T167.

[25] M. Hu, H. Li, Q. Wu, and G. S. Rose, "Hardware realization of bsb recall function using memristor crossbar arrays," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 498–503.

[26] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, R. S. Williams, and J. Yang, "Dot-product engine for neuromorphic computing: programming 1t1m crossbar to accelerate matrix-vector multiplication," in *Proceedings of DAC*, vol. 53, 2016.

[27] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European*

Conference on Computer Systems. ACM, 2013, pp. 169–182.

[28] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: large-scale graph computation on just a pc," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 31–46.

[29] T. Lahiri, M.-A. Neimat, and S. Folkman, "Oracle timesten: An in-memory database for enterprise applications." *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 6–13, 2013.

[30] M. LeBeane, S. Song, R. Panda, J. H. Ryoo, and L. K. John, "Data partitioning strategies for graph workloads on heterogeneous clusters," in *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2015, pp. 1–12.

[31] M.-J. Lee, C. B. Lee, D. Lee, S. R. Lee, M. Chang, J. H. Hur, Y.-B. Kim, C.-J. Kim, D. H. Seo, S. Seo *et al.*, "A fast, high-endurance and scalable non-volatile memory device made from asymmetric ta2o5-x/tao2- x bilayer structures," *Nature materials*, vol. 10, no. 8, pp. 625–630, 2011.

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

[33] G. Linden, B. Smith, and J. York, "Amazon. com recommendations: Item-to-item collaborative filtering," *IEEE Internet computing*, vol. 7, no. 1, pp. 76–80, 2003.

[34] T.-y. Liu, T. H. Yan, R. Scheuerlein, Y. Chen, J. K. Lee, G. Balakrishnan, G. Yee, H. Zhang, A. Yap, J. Ouyang *et al.*, "A 130.7-2-layer 32-gb reram memory device in 24-nm technology," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 1, pp. 140–153, 2014.

[35] X. Liu, M. Mao, B. Liu, H. Li, Y. Chen, B. Li, Y. Wang, H. Jiang, M. Barnell, Q. Wu *et al.*, "Reno: A high-efficient reconfigurable neuromorphic computing accelerator design," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE.* IEEE, 2015, pp. 1–6.

[36] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[37] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[38] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2016, pp. 14–26.

[39] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data.* ACM, 2010, pp. 135–146.

[40] R. Mihalcea and D. Radev, *Graph-based natural language processing and information retrieval.* Cambridge University Press, 2011.

[41] B. Murmann, "Adc performance survey 1997-2016," http://web.stanford.edu/~murmann/adcsurvey.html.

[42] K. P. Murphy, *Machine learning: a probabilistic perspective.* MIT press, 2012.

[43] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on.* IEEE, 2017, pp. 457–468.

[44] D. Niu, C. Xu, N. Muralimanohar, N. P. Jouppi, and Y. Xie, "Design of cross-point metal-oxide reram emphasizing reliability and cost," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE, 2013, pp. 17–23.

[45] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles.* ACM, 2011, pp. 29–41.

[46] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on.* IEEE, 2016, pp. 166–177.

[47] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[48] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *Hot Chips*, vol. 23, 2011.

[49] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 2009, pp. 14–23.

[50] M. Randles, D. Lamb, and A. Taleb-Bendiab, "A comparative study into distributed load balancing algorithms for cloud computing," in *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on.* IEEE, 2010, pp. 551–556.

[51] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. [Online]. Available: http://networkrepository.com

[52] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* ACM, 2013, pp. 472–488.

[53] D. Sanchez and C. Kozyrakis, "Zsim: fast and accurate microarchitectural simulation of thousand-core systems," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 475–486.

[54] J. B. Schafer, D. Frankowski, J. Herlocker, and S. Sen, "Collaborative filtering recommender systems," in *The adaptive web.* Springer, 2007, pp. 291–324.

[55] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. ISCA*, 2016.

[56] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined ReRAM-based accelerator for deep learning," in *High Performance Computer Architecture (HPCA), 2017 IEEE 23rd International Symposium on.* IEEE, 2017.

[57] S. Song, M. Li, X. Zheng, M. LeBeane, J. H. Ryoo, R. Panda, A. Gerstlauer, and L. K. John, "Proxy-guided load balancing of graph processing workloads on heterogeneous clusters," in *2016 45th International Conference on Parallel Processing (ICPP)*, Aug 2016, pp. 77–86.

[58] S. Song, X. Zheng, A. Gerstlauer, and L. K. John, "Fine-grained power analysis of emerging graph processing workloads for cloud operations management," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 2121–2126.

[59] G. Vigna and R. A. Kemmerer, "Netstat: A network-based intrusion detection approach," in *Computer Security Applications Conference, 1998. Proceedings. 14th Annual.* IEEE, 1998, pp. 25–34.

[60] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic i/o optimization for disk-based graph processing," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[61] M. J. Wainwright and M. I. Jordan, "Graphical models, exponential families, and variational inference," *Foundations and Trends in Machine Learning*, vol. 1, no. 1-2, pp. 1–305, 2008.

[62] F. E. Walter, S. Battiston, and F. Schweitzer, "A model of a trust-based recommendation system on a social network," *Autonomous Agents and Multi-Agent Systems*, vol. 16, no. 1, pp. 57–74, 2008.

[63] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan, "Replication-based fault-tolerance for large-scale graph processing," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* IEEE, 2014, pp. 562–573.

[64] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, 2016, p. 11. [Online]. Available: https://github.com/gunrock/gunrock

[65] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, "Metal–oxide rram," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[66] X. Xie, W. Tan, L. L. Fong, and Y. Liang, "Cumf_sgd: Fast and scalable matrix factorization," *arXiv preprint arXiv:1610.05838*, 2016. [Online]. Available: https://github.com/cuMF/cumf_sgd

[67] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems.* ACM, 2013, p. 2.

[68] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2015, pp. 476–488.

[69] Y. Zhao, K. Yoshigoe, M. Xie, S. Zhou, R. Seker, and J. Bian, "Lightgraph: Lighten communication in distributed graph-parallel processing," in *2014 IEEE International Congress on Big Data.* IEEE, 2014, pp. 717–724.

[70] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 375–386.