GaaS-X: Graph Analytics Accelerator Supporting Sparse Data Representation using Crossbar Architectures

Nagadastagiri Challapalle[1*] Sahithi Rampalli[1*] Linghao Song[2] Nandhini Chandramoorthy[3]
Karthik Swaminathan[3] John Sampson[1] Yiran Chen[2] Vijaykrishnan Narayanan[1]
(1) The Pennsylvania State University (2) Duke University (3) IBM T.J. Watson Research Center
{nrc53, svr46, jms1257, vxn9 }@psu.edu {linghao.song, yiran.chen}@duke.edu,
{Nandhini.Chandramoorthy, kvswamin}@us.ibm.com

***Abstract*—Graph analytics applications are ubiquitous in this era of a connected world. These applications have very low compute to byte-transferred ratios and exhibit poor locality, which limits their computational efficiency on general purpose computing systems. Conventional hardware accelerators employ custom dataflow and memory hierarchy organization to overcome these challenges. Processing-in-memory (PIM) accelerators leverage massively parallel compute capable memory arrays to perform the in-situ operations on graph data or employ custom compute elements near the memory to leverage larger internal bandwidths. In this work, we present GaaS-X, a graph analytics accelerator that inherently supports the sparse graph data representations using an in-situ compute-enabled crossbar memory architectures. We alleviate the overheads of redundant writes, sparse to dense conversions, and redundant computations on the invalid edges that are present in the state of the art crossbar-based PIM accelerators. GaaS-X achieves 7.7× and 2.4× performance and 22× and 5.7×, energy savings, respectively, over two state-of-the-art crossbar accelerators and offers orders of magnitude improvements over GPU and CPU solutions.**

***Index Terms*—processing-in-memory, crossbar memory , sparsity, SpMV, graph processing**

I. Introduction

Graphs are powerful and versatile data structures that efficiently model relationships and processes in social, biological, physical and information systems. Graph analytics, or graph algorithms, leverage the graphs to uncover the direction and strength of relationships among the connected nodes (such as objects or people). In the current era of Big-Data and the Internet of Things, understanding the relationships among massive collections of data items plays a vital role for many applications leading to the widespread adoption of graph algorithms. Graph algorithms are widely used in transportation systems, recommendation systems, machine learning, biology, fraud detection and network analysis.

The efficiency of general purpose computing systems for graph algorithms on large-scale, real world graphs is limited by their random and irregular communication patterns [4]. Graph algorithms typically have lower compute-to-communication ratios, processing large numbers of vertices or edges with only a small amount of computation per vertex or edge. Owing to the iterative and sequential nature of the algorithms, reuse of vertex or edge data is limited, leading to poor temporal locality. The random and scale free nature of connectivity in real-world graphs leads to poor spatial locality and load imbalance [28]. Several software frameworks have been proposed to improve the efficiency of large scale processing in both distributed [12], [21], [22], [38] and single-node computing [8], [17], [18], [31], [35], [46] systems using either vertex/edge-centric or sparse matrix vector multiplication (SpMV) execution models. The distributed frameworks partition the graph across different computation nodes and perform computations in parallel across the nodes. The single-node frameworks carefully orchestrate the data layout of the graph (partitioning the graph into grids, shards etc.) to improve the locality of the graph data in memory and allow parallel execution without significant synchronization overheads. Despite the dedicated software frameworks, the graph algorithms inherently suffer from memory access bottlenecks, synchronization and load balancing in both CPU and GPU systems [4], [28], [42].

Owing to the ever increasing importance of graph analytics and demand for their efficient execution, several hardware accelerators have been proposed. These can be broadly categorized into conventional von Neumann architecture-based digital accelerators [13], [28] and processing-in-memory (PIM) accelerators [1], [20], [33], [44]. The digital accelerators implement custom execution pipeline and memory organizations to support the irregular accesses and synchronization overheads in graph processing. In contrast, PIM accelerators leverage the large internal bandwidth of the memory systems and in-memory operations to accelerate graph algorithms by moving the computation closer to the memory to alleviate the data movement bottlenecks in conventional accelerators. Inherently, graph data can be represented as a sparse matrix and several computations in graph processing can be decomposed into SpMV operations. The dominant approach used for graph processing on ReRAM-based PIM accelerators is to convert

*Authors contributed equally.

This work was supported in part by Semiconductor Research Corporation (SRC) Center for Brain-inspired Computing Enabling Autonomous Intelligence (C-BRIC) and Center for Research in Intelligent Storage and Processing in Memory (CRISP).

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00044

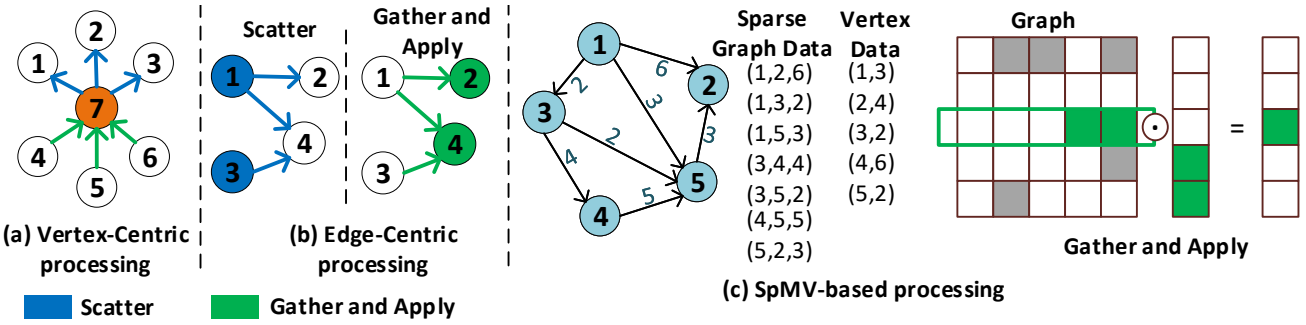

Fig. 1. (a) Vertex-centric programming (b) Edge-centric programming and (c) Sparse Matrix Vector based computations for graphs

a sparse graph representation into a dense representation and map it to parallel MAC operations of the ReRAM crossbar arrays [33]. Although, this approach achieves significant performance improvements over software frameworks and other hardware accelerators, it incurs overheads in sparse-to-dense format conversion, redundant computations on zero valued edges, and additional storage.

In this paper, we present GaaS-X, a crossbar-based PIM accelerator architecture for graph analytics applications. In contrast to the existing PIM crossbar architectures [33], we support computation directly on sparse data representations, alleviating the sparse-to-dense conversions and redundant storage of graph data and computations on zero valued edges. In addition to crossbar MAC operations, GaaS-X leverages emerging nonvolatile memory (NVM) organizations as Content Addressable Memories (CAMs) to perform the in-situ computation on sparse data.

The key contributions of this paper include:

- We present GaaS-X, a versatile crossbar-based in-situ accelerator architecture that efficiently adapts the SpMV computation model to different graph algorithms (e.g. traversal, machine learning) with inherent support for sparse data representation. We incorporate CAM operations to efficiently map SpMV computation to crossbar-based MAC operations on sparse data.
- We demonstrate that CAM-enabled in-situ processing of sparse data achieves a 30x reduction in write operations and a 20x reduction in computations with respect to dense mapping.
- We perform a detailed analysis of performance and energy trade-offs/benefits in mapping several fundamental graph algorithms to our accelerator architecture. We compare state-of-the-art software frameworks on both CPU, GPU and PIM based accelerator architectures to our architecture. GaaS-X achieves 7.7×, 2.4×, 12×, 805× performance and 22×, 5.7×, 252×, 5357× energy savings over the state of the art crossbar accelerators GraphR [33], GRAM [44] and GPU (Gunrock [39], cuMF [37]), CPU (GridGraph [46], GraphChi [18]) respectively.

II. Background & Motivation

A. Graph Processing

The ubiquity of graph data structures in various applications has lead to several efforts in efficient parallel processing of large graphs and optimizations to ensure sequential access to graph data. Some of the initial efforts focused on distributed processing on partitioned graphs. Pregel [22] introduced a computational model known as "Think Like a Vertex" to realize a broad set of graph algorithms. In this vertex-centric model, graph processing can be expressed as a sequence of supersteps. In each superstep a vertex receives messages, processes them to modify its own state and the state of the outgoing edges by passing messages across the outgoing edges (shown in Figure 1(a)). However, this model incurs a large amount of random accesses given that the modified set of vertices at the end of an iteration may not be contiguous. As illustrated in Figure 1, this model can be mapped to three main operations: gather-apply-scatter (GAS) [12], [18].

To ameliorate the cost of random accesses, X-Stream [31] proposes an edge-centric programming paradigm. In this paradigm, edges are streamed from disk storage to avoid random accesses in the edge set and the gather-apply-scatter (GAS) phases are performed on the edges instead of the vertices as illustrated in Figure 1(b). Although this model incurs the cost of random accesses into the vertex set, it achieves significant performance benefits over a vertex-centric model as the edge set is usually much larger than the vertex set. GraphMat [35] maps vertex-centric programs to high performance sparse matrix vector multiplication operations, thereby achieving both the productivity benefits of vertex-programming and high performance of a matrix backend. It leverages high performance computing techniques for sparse linear algebra. Figure 1(c) shows how the GAS programming model can be mapped to SpMV computations.

B. Graph Data Representation

Several solutions have been proposed to address the locality problem of graph processing systems on single machines [8], [17], [18], [46]. Intuitively, large graph data is represented in sparse formats to avoid wasted storage for non-existing relations, some of the popular formats being coordinate list

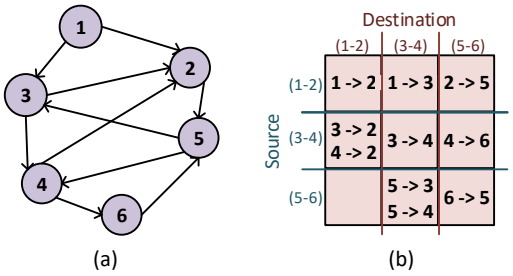

Fig. 2. (a) Sample graph (b) Graph representation on disk

(COO), compressed sparse row (CSR) and compressed sparse column (CSC). The goal of single system disk based graph processing is to partition the graph data into grids [46] or sub-shards [8] in such a way that random accesses to the disk are minimized. They partition the vertex set into disjoint intervals of predefined fixed size. The edges corresponding to a pair of intervals form a sub-shard or a grid and will be stored in a contiguous manner to increase the locality. Figure 2 shows a sample graph and its representation when the interval size is 2. The graph is processed by loading the vertices/edges from each sub-shard onto the memory. Several frameworks also load many sub-shards into the memory at a time and process them in parallel. GaaS-X also employs similar storage mechanism for storing the graph data. We process several sub-shards in parallel, either in column or row major direction depending on the algorithm.

C. Graph Processing on Crossbars

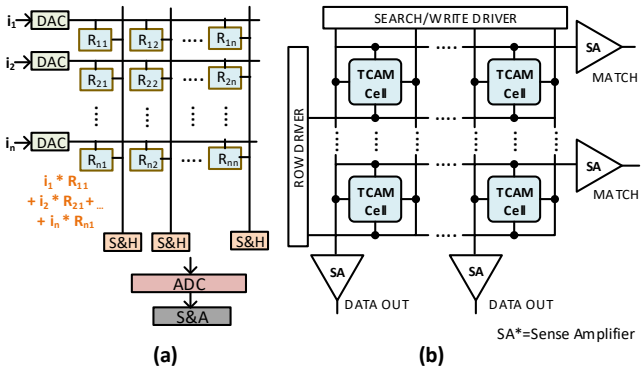

Fig. 3. (a) Dot product operation in crossbar memory and (b) Ternary content addressable memory overview.

ReRAM-based crossbar arrays can perform many parallel in-memory MAC operations [7], [32], [34], [43]. An input voltage applied to the crossbar cells is converted to current by using appropriately configured cell resistors ($I = V \times R$). Further, in accordance with Kirchoff's law, the current from each cell is accumulated along the bit-lines of the crossbar, thus performing a sum of products (MAC) operation as shown in Figure 3(a). The resulting current is sensed and held in sample and hold (S&H) units. The digital inputs can be fed to the crossbar cells with the help of digital to analog converters (DACs) and the digital outputs can be obtained with the help of an analog to digital converter (ADC). A dense matrix vector multiplication operation can also be realized on a crossbar in a similar manner. Vertex programming can be mapped to sparse matrix vector multiplications and additions using the SpMV graph processing model. The state-of-the-art ReRAM based graph accelerator, GraphR [33], fuses the SpMV model with the ReRAM crossbar MAC operation to design an efficient processing-in-memory accelerator for graph applications. GraphR partitions the entire graph, which is in COO format, into subgraphs. In each execution step, a subgraph is loaded into the ReRAM memory, each block of edges in sparse format is converted into a dense matrix, as shown in Figure 4(a), and the dense matrix vector multiplication is performed on the crossbar as discussed above. Figure 4(b) illustrates the computation performed on the compute ReRAMs. Each dense matrix mapped onto a crossbar is equivalent to a sub-block of the graph's adjacency matrix. In the ideal scenario, the non-existent edges or edges with zero weights should not be processed. In contrast, due to the nature of parallel MAC operations in a crossbar, GraphR processes or performs redundant computations on non-existent edges (edges with zero weights in the adjacency matrix). The GraphR [33] architecture can perform close to the ideal case for graph workloads which have a combination of densely populated sub-blocks and zero-edge sub-blocks as it avoids loading of zero-edge sub blocks and any associated computation. However, our analysis on several real-world representative graph workloads showed that 90% of the non-zero sub-blocks have only 10% density. GraphR inevitably performs several additional computations per block due to the dense matrix representation of sparse data, as illustrated in Figure 4(b). Moreover, it always incurs the overhead of converting from the sparse representation in memory ReRAMs to a dense representation in compute ReRAMs.

Figure 5 shows the overhead of write operations and redundant computations incurred due to the sparse-to-dense conversions and parallel computations on dense data normalized to the ideal case for various graph datasets (listed in Table II). The overhead of redundant write operations for each graph dataset is averaged over the graph algorithms PageRank and SSSP. We can observe that, on average, across all the workloads, dense mapping (with a tile size of 16×16) incurs 34× more write operations and 23× more computations than sparse mapping. In addition to the widely incorporated MAC operation, ReRAM crossbar arrays are also capable of content based search operation. Content addressable memories support search of data among the contents of a stored array. Ternary content addressable memories (TCAMs) have additional capabilities for producing a match while ignoring certain bit positions. This enables masking certain bit patterns during the search operation. Figure 3(b) shows an ReRAM based TCAM organization [16] where each cell contains two complementary memory elements to represent one bit of data. By applying complementary bias voltages across the ReRAM elements, we

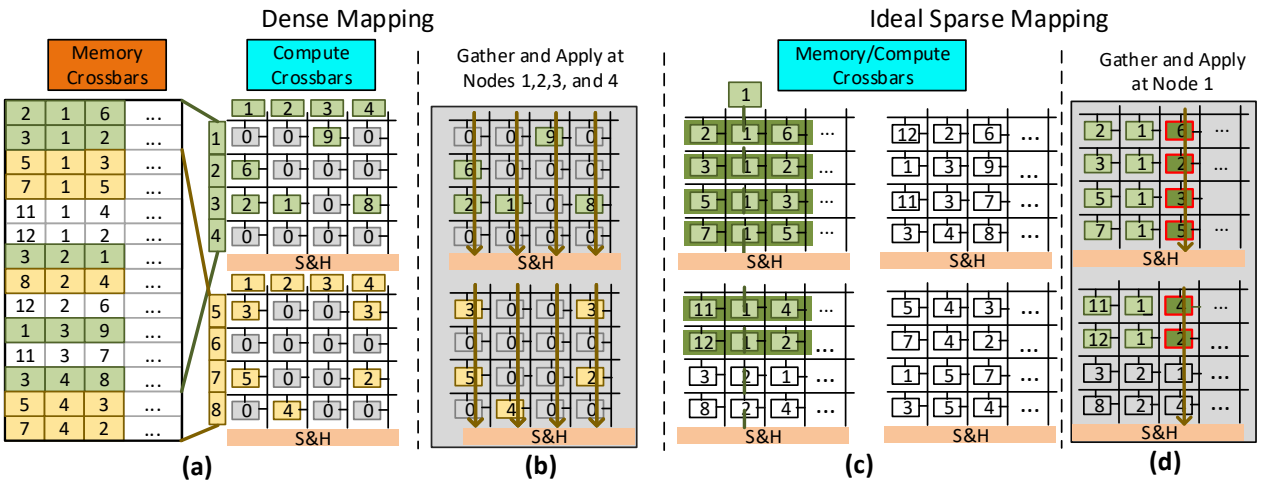

Fig. 4. (a) Mapping sparse graph data in Memory ReRAMs to Compute ReRAMs by converting into dense format (b) Basic operation in Compute ReRAMs of GraphR (c) Ideal mapping of sparse graph data to Compute ReRAMs and (d) Gather and Apply operation on a node performed in Compute ReRAMs with sparse mapping

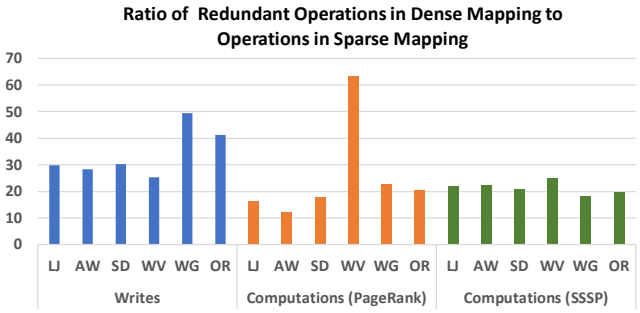

Fig. 5. Ratio of redundant writes and computations in dense mapping normalized with respect to the sparse mapping for the graph datasets listed in Table II

can achieve a bitwise XNOR-based search operation across each cell. In this way, the search key can be broadcast across multiple rows enabling parallel associative matches. When the search pattern matches the stored array contents from a row, the corresponding sense amplifier from the row signals a match, which would be captured for further processing.

D. Graph Processing on GaaS-X

In an ideal ReRAM-based graph processing model, it is possible to alleviate redundant computations on the zero valued edges and sparse-to-dense conversions by mapping the graph data in sparse format to the crossbars as illustrated in Figure 4(c). The ideal mapping enables in-situ processing of sparse data in crossbars without any computation on zero edge weights. Figure 4(d) illustrates a sample *gather* operation on all the edges with destination node 1 and apply the result on node 1. We accumulate only the edge values corresponding to destination 1 (crossbar cells highlighted in red). However, to enable the selective accumulation of the values, we should be able to identify the rows corresponding to the source or destination vertex of interest as shown in Figure 4(c) (crossbar cells highlighted in green). Therefore, in this work, we design the GaaS-X architecture that *leverages the CAM functionality of the crossbar arrays in addition to the MAC functionality to perform in-situ sparse computation using selective accumulation, thereby eliminating the overheads of sparse-to-dense conversions typically incurred in the form of redundant write and compute operations*.

In contrast to the dense mapping, this ideal mapping scenario in GaaS-X supports gather and apply operations only on a single vertex per crossbar array. However, owing to the extreme sparsity in the real world scale-free graphs, it avoids several redundant computations on zero valued edges and also additional writes and storage overheads incurred during sparse-to-dense conversions.

III. GaaS-X Overview

GaaS-X is a processing-in-memory hardware accelerator architecture for the graph analytics algorithms. GaaS-X leverages the content addressable (CAM) and in-situ analog multiply-and-accumulate (MAC) capabilities of the crossbar memory structures to efficiently map the graph algorithms. It transparently supports the native sparse representation of graph data without further preprocessing. It efficiently transforms the process-edge (SpMV-multiply) and reduce (SpMV-add) operations in the SpMV graph computation model [33], [35] to crossbar CAM and MAC operations.

A. Architecture

The GaaS-X architecture comprises four primary components: 1. Central controller, 2. Crossbar memory arrays with CAM functionality (CAM crossbars), 3. Crossbar memory arrays with MAC functionality (MAC crossbars), 4. Special Function Units (SFUs). Figure 6 provides an overview of the GaaS-X architecture and the organization of several components. The central controller is responsible for loading the

graph data into the compute memory structures and attribute buffers. It loads the source and destination pairs of edges onto CAM crossbar arrays and edge attributes such as weights, ranks, and feature vectors onto either MAC crossbar arrays or storage buffers depending on the graph algorithm. It also generates the necessary control logic to the compute crossbar arrays and SFU for the execution of specific graph algorithms. It uses graph meta data such as the vertex ranges loaded into each CAM crossbar, and the particular graph algorithm to generate the required control logic. The CAM crossbars have capabilities to perform parallel searches for a specific data and generate a hit vector (bit map identifying the rows with matches). The MAC crossbars have the capability to perform the MAC operation selectively on data elements either row wise or column wise (i.e. they are transposable crossbars [29]). The SFUs consist of shift and add units (SA) and scalar arithmetic and logic units (sALU) to further process the MAC crossbar outputs and to perform additional operations to support various graph algorithms. The fundamental compute

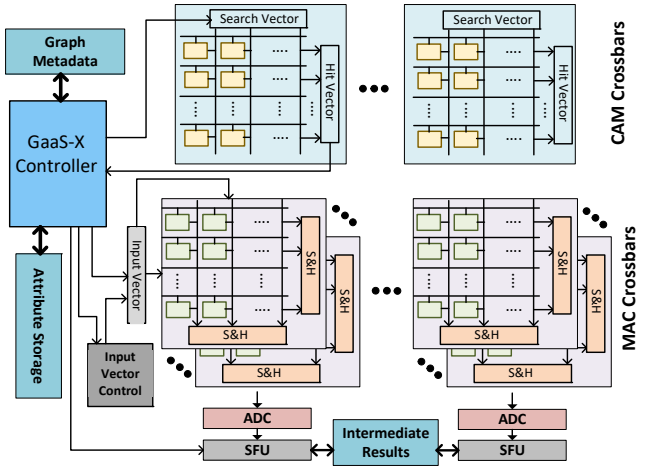

Fig. 6. GaaS-X architecture overview

operation in GaaS-X involves a series of CAM and MAC operations. Consider a sample graph and its coordinate list representation shown in Figure 7(a). The source and destination vertex pairs for each edge are loaded onto CAM crossbars and the edge attributes (i.e. edge weight) are loaded onto MAC crossbars as shown in Figure 7(b). Consider a sample kernel, where all the weights of the incoming edges of a given vertex have to be accumulated. First, we perform a CAM operation to search for the rows which store the given vertex in the destination field. We feed the hit vector from the CAM operation result to the input vector control unit of the MAC crossbar storing the edge weights to selectively enable the rows corresponding to the hit vector. The hit vector only activates the rows containing the weight corresponding to the desired edges, leading to the accumulation of those weights during MAC operation (Figure 7(b)). We store several source-destination vertex pairs and the corresponding edge attributes in CAM and MAC crossbars, respectively, to utilize their storage capabilities in full. However, we accumulate only 16 values in each MAC operation to reduce the peripheral overheads.

The operations in an SpMV computation model, such as parallel matrix multiply and parallel add operations [33], can be decomposed into a series of GaaS-X CAM and MAC operations. For certain graph algorithms, vertex attributes, instead of edge attributes, need to be accumulated. For these, we read the nodes corresponding to the hit vector and use them to selectively enable the rows/columns in the MAC crossbars. In contrast to the mapping of the graph data in dense format to the compute crossbars, GaaS-X maps the graph data in sparse format. Mapping graph data in sparse format avoids the conversion from sparse-to-dense format, redundant writes to storage crossbars, and also many unnecessary compute operations on zero valued elements. Even though the dense mapping approach could result in parallel execution of an entire tile (8x8 or 16x16), there would be redundant computations owing to the extreme sparsity in graph data.

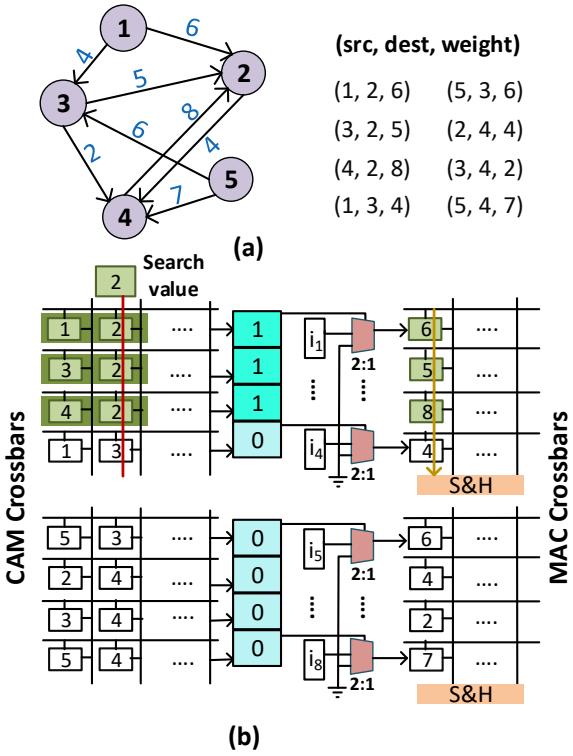

Fig. 7. GaaS-X computing operation

B. Execution Model

The GaaS-X execution model comprises five key phases: 1. Initialization, 2: Data loading, 3: CAM search, 4: MAC operation, 5: Special function execution. Figure 8 gives the overview of the execution model. In the initialization phase, the GaaS-X central controller stores the parameters of the given graph algorithm and the necessary metadata related to the given graph in the Graph Metadata buffer. In the

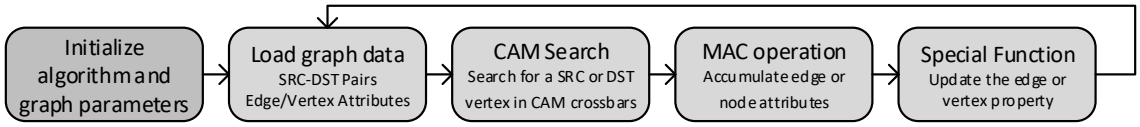

Fig. 8. GaaS-X execution flow

data loading phase, the central controller loads the source-destination pairs of the graph edges onto CAM crossbars, edge attributes or weights onto MAC crossbars. Depending on the given graph algorithm it also stores the relevant vertex attributes, such as embedding, rank, or distance vectors, onto MAC crossbars and on-chip storage buffers. Depending on the graph algorithm, we employ different bit interleaving schemes to support parallel SpMV multiply and add operations on either edge or vertex attributes. The central controller loads a fixed number of source-destination pairs and edge attributes depending on the hardware configuration of GaaS-X (dimension and number of crossbars, precision of source and destination vertices, etc.). It also keeps track of the range of source and destination vertices loaded onto individual crossbars. In the current work, we assume the graph data on the disk is partitioned into sub-shards with each sub-shard associated with a source and destination interval. We adapt this model from the single machine frameworks such as GridGraph [46], GraphChi [18] and NXGraph [8]. Without loss of generality, we assume the edges within a sub-shard are sorted by destination vertices. The shards are loaded in the increasing order of either source interval (row-wise) or destination interval (column-wise) depending on the suitability for the algorithm resulting in sequential disk accesses.

In the CAM search phase, the central controller searches for a particular source or destination vertex within the loaded vertex ranges in the CAM crossbars. This phase identifies the rows/columns to be processed in the MAC operation phase using the resultant hit vector. In the MAC operation phase, the central controller feeds the required input vectors to the crossbars and also selectively enables the rows/columns to be accumulated based on the output of the CAM search phase. The direction of the MAC operation varies depending on the graph algorithm. We employ the necessary logic to achieve this configurability in the peripherals of the MAC crossbars. Note that, irrespective of sparse or dense mapping of graph data to MAC crossbars, this configurability is necessary to support graph algorithms processing vertex attributes such as collaborative filtering and graph neural networks.

In the special function execution phase, the outputs from the MAC operation phase are further processed based on the given graph algorithm. For example, in the graph traversal algorithms, the distance vector update requires calculation of the minimum among the probable distances from the accumulations from different crossbars. Finally, the vertex or edge properties are updated to the on-chip storage buffers using the central controller, depending on the given algorithm. The on-chip storage is large enough to store all the attributes of the vertices loaded onto the crossbars in an execution cycle. Hence, the random accesses to update the vertex attributes are localized to the on-chip storage. The special function unit consists of scalar compute components such as adders, comparators and multipliers. It also consists of required storage buffers and control logic.

IV. Algorithm Mapping

In this section, we provide a generalized procedure to map the SpMV operations (SpMV-multiply and SpMV-add) [35] in the graph processing algorithms to the GaaS-X architecture. We also illustrate the mapping of algorithms representative of graph traversal (SSSP, BFS), matrix-vector multiplication (PageRank (PR)) and machine learning (Collaborative Filtering (CF)) domains to the GaaS-X architecture.

SpMV mapping to GaaS-X: The SpMV-multiply and add operations aggregate or process the vertex/edge attributes (analogous to messages in vertex centric programming) at the vertices depending on the graph algorithm. The SpMV-multiply is used to perform parallel aggregation of vertex/edge attributes at a given set of vertices (such as in PR). The SpMV-add is used to perform parallel add operations to update the attributes of neighbouring vertices of an active vertex (such as in BFS, SSSP). GaaS-X provides the flexibility to aggregate the vertex/edge attributes at source or destination vertices and to selectively update the attributes of neighboring vertices. The ternary CAM operation enables the flexibility to identify the edges corresponding to a particular source or destination vertex. MAC crossbars are equipped with the peripheral circuitry to enable the MAC operation across either row or column. In the following sections, we illustrate mapping of SpMV-multiply operation (PR, CF) and SpMV-add operation (SSSP, BFS, CF).

Single Source Shortest Path (SSSP): The SSSP algorithms are widely used in applications such as route maps, robotics and VLSI design. In this algorithm, a vertex in a directed weighted graph is designated as the start vertex and the distances ($dist$) of all other vertices from the start vertex are computed. The distance of each vertex is updated according to Equation (1).

$$dist(V) = \min_{U:(U,V)\in E}(dist(V), dist(U) + E_{weight}(U,V)) \quad (1)$$

An illustration of SSSP mapped to GaaS-X for a sample graph (Figure 9(a)) is shown in Figure 9(b). In each data loading phase, the source-destination pairs ((U,V)) and the edge weights (E_{weight}) are loaded into the CAM crossbars and

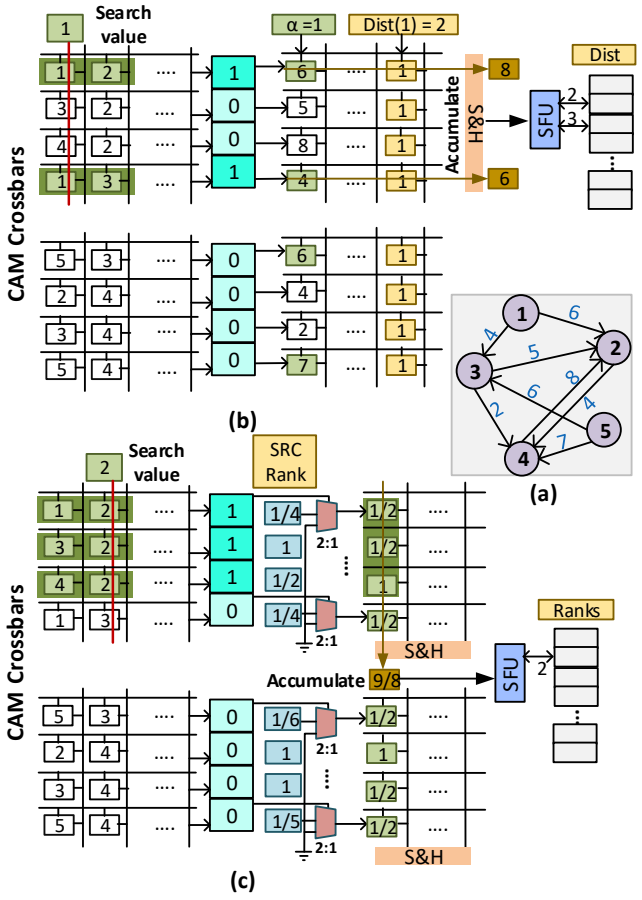

Fig. 9. (a) Sample Graph (b) SSSP and (c) Pagerank

the MAC crossbars respectively. The cells of the last column of MAC crossbars are set to value 1.

In the CAM search phase, we search for the source-destination pairs corresponding to a particular source vertex from the source vertex range of each crossbar. The resultant hit vectors are sent to the corresponding input vector control units of MAC crossbars. The MAC crossbars perform the SpMV-add operation according to the equation, $\alpha \times E_{weight}(U,V) + dist(U) \times 1$ where $\alpha = 1$. This is achieved by feeding α as the input to the edge weights column and the current distance of the chosen source ($dist(U)$) as the input to the last column storing values as 1 as shown in Figure 9(b). The summation is done only on the rows that are enabled by the hit vector. Finally, the distances of the corresponding destination vertices of the enabled edges are updated using the minimum function from the special function units. The destination vertices can be obtained from the enabled CAM rows. This process is repeated for all the source vertices in the current range before loading the next set of edges. Thus, the distances of the destination vertices in the current range are updated.

Breadth-First Search (BFS): BFS is a path-finding or path-counting problem which can be formulated mathematically according to equation (2).

$$dist(V) = \min_{U:(U,V)\in E}(dist(V), dist(U)+1) \tag{2}$$

Note that this equation is similar to SSSP's basic compute step with edge weights fixed to a value of one. Thus, BFS can be performed similar to SSSP without the overhead of loading edge weights into MAC crossbars but setting the edge weight columns to a fixed value of 1.

PageRank (PR): The PageRank algorithm is one of the methods used by Google search to determine the popularity (rank) of a webpage. A webpage (vertex) is ranked based on the ranks of the neighbouring webpages. A rank of a vertex is equally split among all the neighbours along the outgoing edges ($OutDeg$) of the vertex. The vertices are ranked according to Equation (3). α is a constant. This problem can be formulated as an eigen decomposition problem and hence, can be solved using SpMV methods [35].

$$rank(V) = (1-\alpha) + \alpha \sum_{U:(U,V)\in E} \frac{rank(U)}{OutDeg(U)} \tag{3}$$

Figure 9(c) shows an example mapping of PageRank to GaaS-X for the sample graph in Figure 9(a). In the data loading phase, the source-destination vertex pairs are loaded into the CAM crossbars and the reciprocals of the out degrees (number of outgoing edges) of the source vertices are loaded into the MAC crossbars. The original ranks of the source vertices and the new/updated ranks of the destination vertices are loaded into the attribute buffers.

In the CAM search phase, each destination vertex from the destination vertex range is chosen and the CAM operation is performed on the destination columns of the CAM crossbars. The operation to be performed in the MAC crossbars is given in Equation 4.

$$\begin{aligned} &\sum_{U:(U,V)\in E} rank(U) * \frac{1}{OutDeg(U)} \\ &\simeq \sum_{U:(U,V)\in E} rank(U) * E_{weight}(U,V) \end{aligned} \tag{4}$$

The hit vector enables the rows of the MAC crossbars that need to be accumulated to complete the SpMV-multiply operation. The input to the crossbars is the ranks of the source vertices ($rank(U)$) of the enabled edges. The MAC operation is performed on the input ranks and the edge weights of the enabled rows. Finally, the rank of the chosen destination vertex is updated in the new rank vector using the SFUs according to Equation (3). This process is repeated for all the destination vertices in the current range before loading the next set of edges. Thus, the ranks of the destination vertices in the current range are updated. Other SpMV based algorithms can also be mapped to GaaS-X architecture in a similar manner as PageRank.

Collaborative Filtering (CF): Collaborative Filtering is widely used in recommendation systems to predict the unknown ratings between user-item pairs and can be used to

recommend an item to a user. The input to CF is an undirected bipartite graph between user and item sets with the rating of a user for an item as the edge weight. Users and items are associated with latent feature vectors. This algorithm uses latent feature vectors of the users (P_u) and the items (P_i) to predict the ratings. Equation (5) shows the main compute steps used in the algorithm. The term e_{ui} is the error in the prediction, G is the adjacency matrix of the users to items with matrix elements as the ratings.

$$
\begin{aligned}
e_{ui} &= G - P_u^T P_i \\
P_u^* &= P_u + \gamma[\sum_{U:(U,I)\in E} (e_{ui} P_i - \lambda P_u)] \\
P_i^* &= P_i + \gamma[\sum_{U:(U,I)\in E} (e_{ui} P_u - \lambda P_i)]
\end{aligned}
\tag{5}
$$

Collaborative filtering can be decomposed into two primary operations, the dot product of the vertex attributes (i.e. user and item feature vectors) $P_u^T P_i$ and the accumulation of vertex attributes based on the presence of an edge between the vertices $\sum_{U:(U,I)\in E}(e_{ui}P_u)$, $\sum_{U:(U,I)\in E}(e_{ui}P_i)$. The rest of the computation can be performed with the help of scalar arithmetic units. The mapping of CF or other machine learning applications is different from the other algorithms discussed above as the main computations involved in these algorithms are performed on vertex properties (such as feature vectors or embedding vectors) rather than edge properties (such as edge weights).

In the data loading phase, the central controller loads the current set of edges, including the edge weights, onto CAM crossbars. The feature vectors of users (P_u) and items (P_i) corresponding to the range of vertex IDs, are loaded into different MAC crossbars. The computations of CF can be divided into two phases: 1. The item update phase corresponding to P_i^* and 2. A user update phase corresponding to P_u^*. Each phase includes both CAM and MAC operations. In the item update phase, we search for each item in the CAM crossbars to identify the corresponding users as shown in Figure 10(b). The corresponding error terms are updated using the dot product operation between the vertices linked through valid edges by performing SpMV-multiply operation on MAC crossbars. In the example shown in Figure 10(b), the feature vector of item 1 is multiplied with feature vectors of users 1, 2 and, 4 as they have given valid ratings to this item. Using these error values, the features of the user vertices which have valid edges to corresponding item are accumulated according to the SpMV-multiply operation on the MAC crossbars, to obtain the updated feature vector for the item as shown in Figure 10(b). During the item update phase, we also update a user list data structure which holds the IDs of the item vertices for which the user has given rating. This user list is further used to accumulate the feature vectors of the items corresponding to each user as shown in the gather operation in Figure 10(c) to update the user feature vectors.

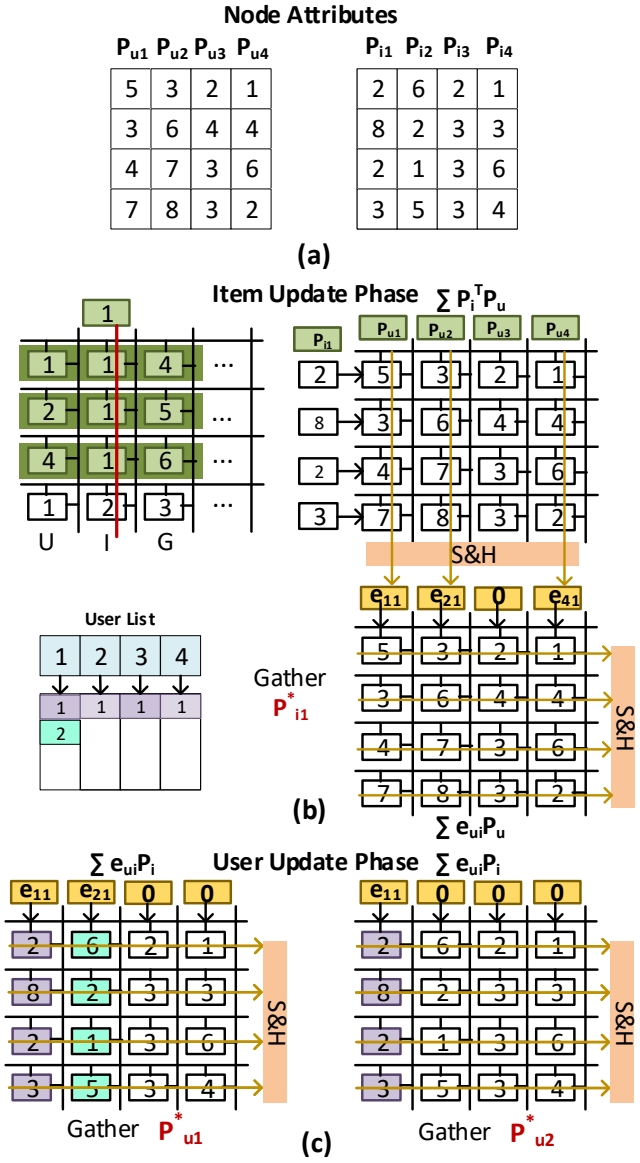

Fig. 10. Collaborative Filtering (a) Attributes of user and item vertices/nodes (b) Item update phase and (c) User update phase.

V. GaaS-X Evaluation

A. Evaluation Methodology

In this section, we discuss the evaluation methodology, benchmarks and comparison baseline architectures used to evaluate GaaS-X.

System Design and Evaluation We have designed a custom cycle accurate simulator for performance and energy efficiency evaluation of GaaS-X. The custom simulator models all the microarchitectural characteristics of the GaaS-X architecture including the on-chip storage buffers, processing-in-memory (PIM) storage and compute elements and other peripheral circuits. The latency and power consumption parameters of the

TABLE I
PARAMETERS OF THE GAAS-X ARCHITECTURE

Component	Configuration	Area ($mm^2 \times 10^{-3}$)	Power (mW)
MAC crossbar	128×16×8 2-bits/cell number: 2048	51.2	307.20
DAC	2-bit number: 256×2048	0.08	1.64
S&H	number: 1152×2048	72.00	2.56
ADC	6-bit, 1.2GSps number: 512	300.80	328.96
CAM Crossbar	128×128 1-bit/cell number: 2048	80.00	614.40
Central Controller		1650.00	50.00
SFU		286.72	33.87
Output Buffer	64 KB	25.60	34.88
Input Buffer	16 KB	6.40	8.72
Attribute Buffer	512 KB	204.80	279.04
Total		**2.69** mm^2	**1.66 W**

on-chip storage buffers (input buffer, output buffer etcetera) are modeled using 32nm CACTI model [3]. The latency and power consumption parameters of the PIM compute elements and other digital compute elements are fed into the simulator for obtaining the overall system level execution time and energy analysis.

We have implemented the overall control circuitry, such as the central controller, multiplexers for the input control and SFU components, in SystemVerilog RTL. The power consumption and area metrics are obtained by doing RTL synthesis in 32nm [36] technology for operation at 1GHz. The latency and power consumption parameters of the CAM crossbar and MAC crossbar arrays are obtained by performing SPICE simulations using the resistive random access memory (RRAM) model from [40] in 32nm technology. We activate only up to 16 wordlines in each compute operation depending on the hit vector from the CAM operation, hence, a 6-bit ADC is sufficient for the accumulation in the MAC operation. We use ADC and DAC models from [11] in our design. The overall latency of MAC operation is 30ns and CAM operation is 4ns. Table I lists the area and power parameters of the various components of our design. The overall area and power consumption of the GaaS-X accelerator is $2.69mm^2$ and $1.66W$ respectively.

Graph Datasets The characteristics of the graph datasets we use for the evaluation are listed in Table II. We ran the PageRank, SSSP and BFS algorithms on the WikiVote (WV) [19], SlashDot (SD) [19], Amazon (AZ) [19], Web-Google (WG) [19], LiveJournal (LJ) [19], and Orkut (OR) [30] datasets. We ran collaborative filtering with a feature size of 32 on the Netflix [6] dataset which consists of user ratings for different movies in the Netflix repository.

Comparison Baselines We have compared the efficiency of GaaS-X with software graph processing frameworks on CPU and GPU. We have used the GridGraph [46] graph processing framework for running PageRank, BFS and SSSP algorithms and the GraphChi [18] framework for running Collaborative Filtering on CPU. We have used the Gunrock [39] framework for running the PageRank, BFS and SSSP algorithms and the cuMF [37] framework for running Collaborative Filtering on GPU. We also compare the performance of PageRank, BFS and SSSP with GAP benchmark suite (GAPBS) [5] which is a highly optimized parallel implementation for graph processing on CPU. The CPU and GPU power consumption are measured using Intel RAPL [15] and Nvidia-SMI [26], respectively. We ensure no user-level process is active during the performance and power analysis. However, since both of these real-system power measurement frameworks incorporate other system effects such as background kernel-level processes beyond the user application into their power measurement, we subtract out measured system idle power before comparing against the power of our accelerator design for a fairer comparison. We also compare GaaS-X with state of the art ReRAM-based analog PIM [33] (GraphR) and digital PIM [44] (GRAM) graph processing accelerators. We simulate the micro architectural characteristics of GraphR (e.g. dense mapping to crossbars) using our custom cycle-accurate simulator with the

TABLE II
GRAPH DATASETS AND CHARACTERISTICS

Dataset	Vertices	Edges	Description
WikiVote (WV)	7.0K	103K	Wikipedia voting data
Slashdot (SD)	82K	948K	Slashdot Zoo social network
Amazon(AZ)	262K	1.2M	Amazon co-purchasing network
WebGoogle (WG)	0.88M	5.1M	Webgraph from Google
LiveJournal (LJ)	4.8M	69M	LiveJournal social network
Orkut (OR)	3.0M	106M	Orkut social network
Netflix (NF)	480K users 17.8K movies	99M	Netflix movie user ratings

TABLE III
BASELINE SYSTEM CONFIGURATIONS

Component	Specification
CPU	Intel(R) Xeon(R) Bronze 3104 Number: 12 Frequency: 1.7 GHz L1 Cache: 12 × (32KB + 32KB) L2 Cache: 12 × (1MB) L3 Cache: 8.25MB
GPU	Nvidia Titan V Architecture: Nvidia Volta Frequency: 1455 MHz Memory: 12 GB HBM2 CUDA Cores: 5120
PIM Accelerators	GraphR [33] GRAM [44]

same technology parameters for the PIM and digital compute elements used to model GaaS-X. We also keep same number of parallel compute elements (2048) in both GaaS-X and GraphR. Since GRAM uses a radically different architecture than the one we model in detail, we only compare with GRAM in terms of the previously reported [44] end-to-end relative performance and energy improvements with respect to GraphR for the AZ, WV and LJ datasets. Table III lists the specifications of the baseline systems used for performance comparison.

B. Evaluation Results

We evaluate the performance of the GaaS-X accelerator on various graph datasets introduced in the previous section.

Comparison with GraphR Figure 11 shows the speedup in execution time of GaaS-X with respect to GraphR [33] for various graph datasets for PageRank, BFS and SSSP algorithms. The geometric mean of execution time speedup across all datasets and algorithms is 7.74. The speedup in execution time of PageRank algorithm is lower than the speed up in BFS and SSSP. The parallelism in GraphR accelerator for PageRank is significantly higher than GaaS-X, as it processes the entire dense graph tile at a time. Although, GraphR performs wasted computations, due to high parallelism, it achieves better throughput for PageRank than the other algorithms. For the BFS and SSSP algorithms, GraphR can process only one row at a time in the graph tile, leading to lower parallelism. Hence, the GaaS-X accelerator shows significant speedup ($\simeq 40$) with respect to GraphR for BFS and SSSP.

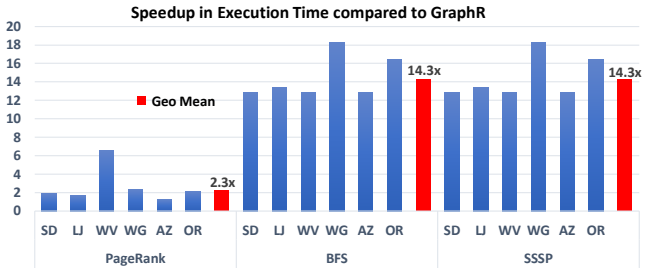

Fig. 11. Speed up in execution time of GaaS-X accelerator for various graph datasets compared to GraphR [33] accelerator.

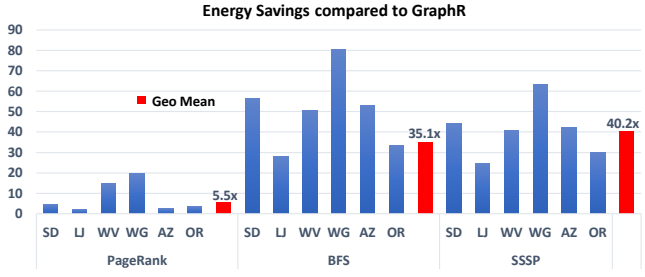

Fig. 12. Energy savings of GaaS-X accelerator for various graph datasets compared to GraphR [33] accelerator.

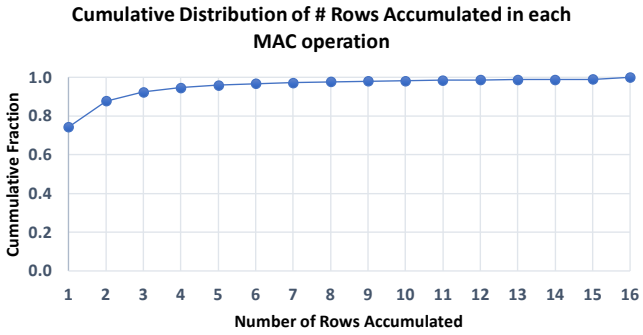

Fig. 13. Fraction of rows being accumulated in each MAC operation across the graph algorithms and datasets.

GraphR, as it does not perform any unnecessary computations on zero edge weights or invalid edges. Also, it does not require any additional writes of the dense graph format to compute ReRAMs. The improvement in energy savings also shows that the additional energy spent in CAM operations is less than the energy consumed in extra writes and the unnecessary computations in GraphR. The number of rows to be accumulated in each MAC operation of GaaS-X varies depending on the hit vector from the CAM search operation. Figure 13 shows the percentage of different accumulated rows (1 to 16) for each of the MAC operations across all the evaluated algorithms and datasets. It can be observed that around 75% of MAC operations accumulate only one row, and the percentage of MAC operations accumulating more than six rows is minimal (3%). The fewer number of rows being accumulated in each MAC operation also contributes to the superior energy savings of GaaS-X. Overall GaaS-X achieves 7.7x speedup and 22x energy savings over GraphR which in turn shows up to 4x performance and 4x-10x energy efficiency gains over Tesseract [1].

Comparison with GRAM Figure 14 shows the comparison of execution time speedup and energy savings of GaaS-X with respect to GRAM [44] for PageRank, BFS and SSSP algorithms. The geometric mean of speedup in execution time and energy savings of GaaS-X are 2.5 and 5.2, respectively. Note that the GRAM architecture is fundamentally different from both GaaS-X and GraphR. It leverages digital PIM operations such as compare-and-swap and parallel reduction operations.

Comparison with GridGraph and Gunrock Figure 15 shows the speedup in execution time of GaaS-X with respect to software graph processing frameworks, GridGraph on CPU and Gunrock, on GPU for the PageRank, BFS and SSSP algorithms. The geometric mean speedups with respect to CPU and GPU across various datasets for the PageRank, BFS and SSSP algorithms are 805.44 and 12.29, respectively. GaaS-X achieves significant speedup compared to the CPU owing to its custom dataflow and parallel in-situ CAM and MAC operations in crossbars. GaaS-X achieves performance gains over the GPU due to the localization of random access

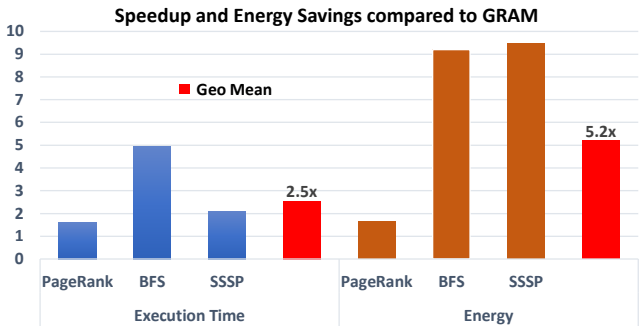

Fig. 14. Performance comparison of GaaS-X with GRAM [44]

for vertex attributes to on-chip storage and mapping the operations to a fusion of low-latency parallel crossbar CAM and MAC operations. GPU framework achieves significant speedup compared to the CPU framework owing to large number of CUDA cores (5120), and the high bandwidth memory (avoids many CPU-managed interactions for data loading/marshalling). Figure 16 shows the energy savings of the GaaS-X accelerator with respect to CPU and GPU for PageRank, BFS and SSSP algorithms. The geometric mean of energy savings across all the datasets and algorithms is 5375 and 252 with respect to CPU and GPU respectively. Significant energy savings are observed in GaaS-X owing to the in-situ processing on low energy ReRAM-based crossbar arrays and custom dataflow tailored primarily for graph processing.

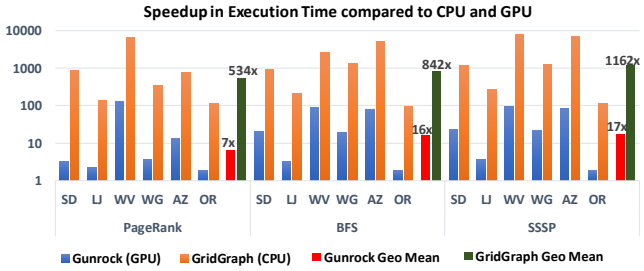

Fig. 15. Speed up in execution time of GaaS-X accelerator for various graph datasets compared to software frameworks on CPU and GPU.

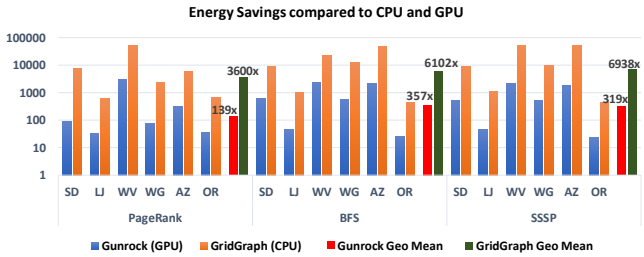

Fig. 16. Energy savings of GaaS-X accelerator for various graph datasets compared to software frameworks on CPU and GPU.

Comparison with GAPBS With respect to GAPBS [5], we observe geometric mean speedup and energy efficiency of $\simeq$155x and $\simeq$1500x, respectively, for PageRank, BFS and SSSP algorithms across all the datasets. The performance and energy improvements can be attributed to the graph processing at shard level, sequential accesses and the low-power analog computations.

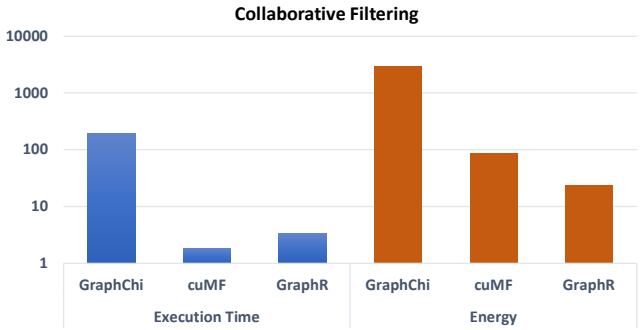

Fig. 17. Comparison performance and energy consumption of GaaS-X with respect to CPU, GPU and GraphR for collaborative filtering.

Collaborative Filtering Figure 17 shows the execution time speedup and energy savings of the GaaS-X accelerator for running collaborative filtering (CF) compared to CPU, GPU and GraphR. We could not compare the performance of GaaS-X on CF algorithm to GRAM [44] as the latter was not evaluated on this algorithm. The execution time speedup and energy savings of GaaS-X with respect to CPU, GPU and GraphR platforms are 196, 2, 4 and 2962, 86, 24 respectively.

The collaborative filtering algorithm processes both edge attributes and vertex attributes of the graph. Note that this execution model is similar to the emerging graph analytics algorithms such as graph neural networks [14], which comprise a series of operations such as accumulation, convolution over vertex attributes and edge attributes. Though these emerging algorithms can be mapped to GaaS-X architecture, in this work, we refrain from this analysis as these algorithms are still evolving and lack standardized benchmarks.

VI. Related Work

A. Software Frameworks

Distributed graph processing software frameworks, such as GraphX [41], Pregel [22], GraphLab [21], and PowerGraph [12], process the large scale graphs by splitting the computations across different nodes and leveraging the large compute resources of clusters. These frameworks provide efficient mechanisms for scheduling and synchronizing the computations across the nodes. Single node software frameworks, such as GraphChi [18], X-Stream [31], TurboGraph [17], GridGraph [46], and NXGraph [8], utilize the disk storage efficiently to incur less random accesses and to enable parallel computations. These frameworks partition the graph data into several disjoint intervals and process them in parallel. They optimize the graph partition such that the vertices or edges corresponding to adjacent intervals reside closer in the disk. GraphMat [35] maps the computations in a vertex programming

model to SpMV operations and accelerates them efficiently using optimized linear algebra solvers. GPU-based software frameworks such as Gunrock [39] and nvGraph [27] leverage the parallel compute cores in GPUs to efficiently schedule the computations in graph algorithms. GraphSSD [23] modifies the conventional logical to physical page mapping with their custom vertex to page mapping scheme to minimize unnecessary page movement overheads. Faldu *et. al.* [10] and Balaji *et. al.* [2] propose custom reordering techniques to preserve the structure of the graphs and to improve the locality.

B. Hardware Frameworks

There are several FPGA implementations specific to acceleration of a particular graph algorithm such as SSSP [45] or PageRank [24] and also for acceleration of the vertex programming paradigm [9] for graph algorithms. Conventional digital accelerators [13], [28] leverage custom dataflows and memory hierarchy organizations to overcome the inefficiencies of software frameworks. They minimize the off-chip memory accesses by efficient scheduling of the operations and usage of the on-chip memory. Mukkara *et. al.* [25] propose a hardware-accelerator for traversal scheduling to exploit the locality in real-world graphs due to their clustered structure. PIM-based accelerators either move the computations closer to the memory and leverage the large internal bandwidth of the memory arrays or leverage the in-situ analog or digital compute capabilities of emerging memory technologies. Tesseract [1] and GraphIA [20] utilize the large internal bandwidth of DRAM arrays to perform parallel operations by instantiating the custom compute units near them. Improving on Tesseract, GraphQ [47], proposes a hybrid execution model to maximize intra and inter-array throughput. GRAM [44] utilizes digital processing-in-memory capabilities such as compare-and-swap and parallel reduction operations of the crossbar memory arrays. GraphR [33] utilizes the extremely parallel and low power analog multiply-and-accumulate capabilities of crossbar memory arrays to map the SpMV model to crossbars. In contrast to the existing DRAM-based PIM accelerators (Tesseract, GraphIA and GraphQ), GaaS-X incorporates in-situ sequential processing of sparse graph data on the ReRAM crossbar memories. In contrast to existing ReRAM-based PIM approaches (GraphR and GRAM), GaaS-X leverages the content addressable capabilities of the crossbar memory arrays to perform in-situ operations directly on the sparse graph data representation.

VII. Conclusion

In this paper, we present GaaS-X, a graph analytics accelerator performing in-situ computations directly on the sparse graph data stored in crossbar memory arrays. We leverage the content-addressable and multiply-and-accumulate (MAC) capabilities of the crossbar memory arrays to efficiently map several representative graph analytics algorithms such as PageRank, SSSP, BFS and collaborative filtering. Utilizing these features of the crossbar memory to perform the operations directly on the sparse data alleviates the need for sparse-to-dense data conversions and redundant computations on invalid edges, and reduces the resulting peripheral overheads incurred in the MAC implementation. This results in significant performance and energy improvements over prior approaches. GaaS-X achieves 7.7× / 2.4× performance and 22× / 5.7×, energy savings over PIM graph accelerators GraphR [33] and GRAM [44] respectively. It achieves 12× / 805× performance and 252× / 5357× energy savings over software frameworks running on GPUs, and CPUs respectively.

Acknowledgements

We thank the anonymous reviewers of ISCA 2020 for their constructive and insightful comments.

References

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 105–117.

[2] V. Balaji and B. Lucia, "When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 203–214.

[3] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 14:1–14:25, 2017.

[4] S. Beamer, K. Asanovic, and D. Patterson, "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server," in *2015 IEEE International Symposium on Workload Characterization*, Oct 2015, pp. 56–65.

[5] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP Benchmark Suite," *CoRR*, 2015.

[6] J. Bennett and S. Lanning, "The Netflix Prize," in *Proceedings of KDD cup and workshop*, 2007.

[7] P. Chi, S. Li, 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 *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 27–39.

[8] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "NXgraph: An efficient graph processing system on a single machine," *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pp. 409–420, 2015.

[9] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 217–226.

[10] P. Faldu, J. Diamond, and B. Grot, "A Closer Look at Lightweight Graph Reordering," in *In Proceedings of the International Symposium on Workload Characterization*, 2019.

[11] D. Fujiki, S. Mahlke, and R. Das, "In-Memory Data Parallel Processor," in *ASPLOS*, 2018, pp. 1–14.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Power-Graph: Distributed Graph-Parallel Computation on Natural Graphs," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, 2012, pp. 17–30.

[13] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–13.

[14] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 1025–1035.

[15] Intel Corporation, "Intel vtune amplifier performance profiler," [online]. Available: https://software.intel.com/en-us/intel-vtune-amplifier-xe, [Accessed: 26- Jun- 2019].

[16] N. Jao, A. K. Ramanathan, A. Sengupta, J. Sampson, and V. Narayanan, "Programmable Non-Volatile Memory Design Featuring Reconfigurable In-Memory Operations," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–5.

[17] S. Ko and W.-S. Han, "TurboGraph++: A Scalable and Fast Graph Analytics System," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 395–410.

[18] 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.

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

[20] G. Li, G. Dai, S. Li, Y. Wang, and Y. Xie, "GraphIA: An In-situ Aelerator for Large-scale Graph Processing," in *Proceedings of the International Symposium on Memory Systems*, 2018, pp. 79–84.

[21] 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," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[22] 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*, 2010, pp. 135–146.

[23] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, "GraphSSD: Graph Semantics Aware SSD," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 116–128.

[24] S. McGettrick, D. Geraghty, and C. McElroy, "An FPGA architecture for the Pagerank eigenvector problem," in *2008 International Conference on Field Programmable Logic and Applications*, 2008, pp. 523–526.

[25] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 1–14.

[26] Nvidia Corporation, "Nvidia system management interface," [online]. Available: https://developer.nvidia.com/nvidia-system-management-interface, [Accessed: 26- Jun- 2019].

[27] Nvidia Corporation, "The NVIDIA Graph Analytics library (nvGRAPH)," [online]. Available: https://developer.nvidia.com/nvgraph, [Accessed: 20- Nov- 2019].

[28] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy Efficient Architecture for Graph Analytics Accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 166–177.

[29] A. Ranjan, S. Jain, J. R. Stevens, D. Das, B. Kaul, and A. Raghunathan, "X-MANN: A Crossbar Based Architecture for Memory Augmented Neural Networks," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 130:1–130:6.

[30] 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, pp. 4292–4293.

[31] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric graph processing using streaming partitions," *SOSP 2013 - Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[32] 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 *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 14–26.

[33] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating Graph Processing Using ReRAM," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 531–543.

[34] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined rerambased accelerator for deep learning," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 541–552.

[35] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High Performance Graph Analytics Made Productive," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, Jul. 2015.

[36] Synopsys, [online]. Available: https://www.synopsys.com/community/university-program/teaching-resources.html, [Accessed: 26- Jun- 2019].

[37] W. Tan, L. Cao, and L. L. Fong, "Faster and Cheaper: Parallelizing Large-Scale Matrix Factorization on GPUs," *CoRR*, vol. abs/1603.03820, 2016. [Online]. Available: http://arxiv.org/abs/1603.03820

[38] The Apache Software Foundation , [online]. Available: https://giraph.apache.org/, [Accessed: 2- Nov- 2019].

[39] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: GPU Graph Analytics," *ACM Trans. Parallel Comput.*, vol. 4, no. 1, pp. 3:1–3:49, Aug. 2017.

[40] Wei Zhao and Yu Cao, "New generation of predictive technology model for sub-45nm design exploration," in *7th International Symposium on Quality Electronic Design (ISQED'06)*, 2006, pp. 6 pp.–590.

[41] 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*, 2013, pp. 2:1–2:6.

[42] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on GPUs: Where are the bottlenecks?" in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2014, pp. 140–149.

[43] T.-H. Yang, H.-Y. Cheng, C.-L. Yang, I.-C. Tseng, H.-W. Hu, H.-S. Chang, and H.-P. Li, "Sparse ReRAM Engine: Joint Exploration of Activation and Weight Sparsity in Compressed Neural Networks," in *ISCA*, 2019, pp. 236–249.

[44] M. Zhou, M. Imani, S. Gupta, Y. Kim, and T. Rosing, "GRAM: Graph Processing in a ReRAM-based Computational Memory," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, pp. 591–596.

[45] S. Zhou, C. Chelmis, and V. K. Prasanna, "Accelerating Large-Scale Single-Source Shortest Path on FPGA," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 129–136.

[46] 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.

[47] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "GraphQ: Scalable PIM-Based Graph Processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, p. 712–725.