Gzippo: Highly-compact Processing-In-Memory Graph Accelerator Alleviating Sparsity and Redundancy

Xing Li
Shanghai Jiao Tong University
Shanghai, China
lixingCIT@sjtu.edu.cn

Rachata Ausavarungnirun
King Mongkut's University of Technology North Bangkok
Bangkok, Thailand
r.ausavarungnirun@gmail.com

Xiao Liu
Bytedance US Lab
Mountain View, CA, USA
xiao.liu@bytedance.com

Xueyuan Liu
Shanghai Jiao Tong University
Shanghai, China
xueyuan_liu@sjtu.edu.cn

Xuan Zhang
Shanghai Jiao Tong University
Shanghai, China
zhangxuan09@sjtu.edu.cn

Heng Lu
Shanghai Jiao Tong University
Shanghai, China
aqluheng@sjtu.edu.cn

Zhuoran Song
Shanghai Jiao Tong University
Shanghai, China
songzhuoran@sjtu.edu.cn

Naifeng Jing
Shanghai Jiao Tong University
Shanghai, China
sjtuj@sjtu.edu.cn

Xiaoyao Liang
Shanghai Jiao Tong University
Shanghai, China
liang-xy@cs.sjtu.edu.cn

ABSTRACT

Graph application plays a significant role in real-world data computation. However, the memory access patterns become the performance bottleneck of the graph applications, which include low compute-to-communication ratio, poor temporal locality, and poor spatial locality. Existing RRAM-based processing-in-memory accelerators reduce the data movements but fail to address both sparsity and redundancy of graph data. In this work, we present *Gzippo*, a highly-compact design that supports graph computation in the compressed sparse format. Gzippo employs a tandem-isomorphic-crossbar architecture both to eliminate redundant searches and sequential indexing during iterations, and to remove sparsity leading to non-effective computation on zero values. Gzippo achieves a 3.0× (up to 17.4×) performance speedup, 23.9× (up to 163.2×) energy efficiency over state-of-the-art RRAM-based PIM accelerator, respectively.

CCS CONCEPTS

• **Computer systems organization → Special purpose systems**.

KEYWORDS

graph computation, processing-in-memory, sparsity, redundancy, RRAM

ACM Reference Format:
Xing Li, Rachata Ausavarungnirun, Xiao Liu, Xueyuan Liu, Xuan Zhang, Heng Lu, Zhuoran Song, Naifeng Jing, and Xiaoyao Liang. 2022. Gzippo: Highly-compact Processing-In-Memory Graph Accelerator Alleviating Sparsity and Redundancy. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, San Diego, CA, USA, 9 pages. https://doi.org/10.1145/3508352.3549372

1 INTRODUCTION

Graph algorithms have been a core component in many modern applications, such as social network, web searching, and travel navigation. Google page rank (PR) algorithm [18] can sort all World Wide Web pages according to the significance. Single source shortest path (SSSP) algorithm [7] contributes to acquisition of the best travel route [30]. However, deploying high-performance graph algorithm still poses significant challenges due to its irregular memory accesses [13] and generally suffers from three major reasons [20]. First, many graph algorithms have low compute-to-communication ratio, leading to more data accesses per actual compute operation for any graph size. Second, graph applications exhibit poor temporal locality due to the irregularity of the graph structure. Third, graph algorithms have poor spatial locality and generate many sparse data accesses over random memory addresses.

Hardware accelerators have been proposed to optimize data scheduling and prefetching based on data dependency and online locality [3, 10, 13, 20]. These hardware mechanisms ameliorate the temporal/spatial locality to reduce the number of miss in cache to improve performance. However, they fail to eliminate the fundamental challenges of graph applications that generate multiple data movements between the memory and the processing core [25].

To further address these challenges, many processing-in-memory (PIM) accelerators are proposed to accelerate graph algorithms [5, 25, 31] by minimizing data movements through in-memory computation. Several proposals exploit the Non-Volatile Memory (NVM) (e.g., resistive random access memory (RRAM)) as substrate to perform graph operations. For example, GraphR [25], implementing a PIM graph accelerator through RRAM, enables the acceleration of graph processing by performing sparse-matrix-vector multiplication (SpMV) on RRAM's crossbar. GraphR can leverage the large

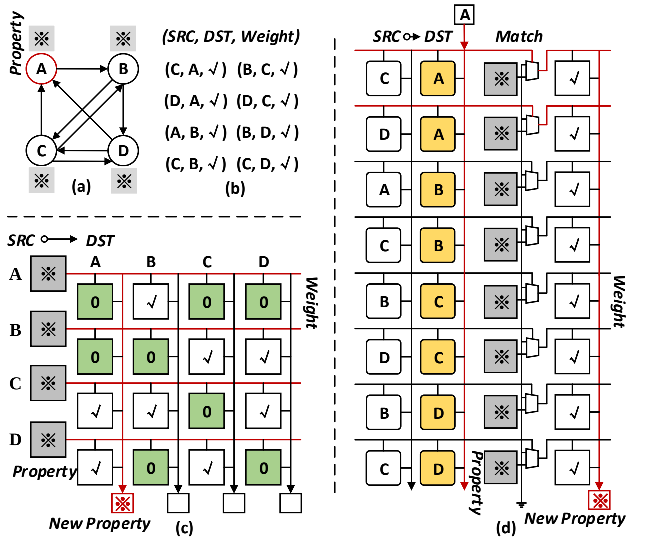

Figure 1: (a) Graph sample (b) Coordinate representation format (c) SpMV-supported scheme with sparsity in weight array (d) CAM-based scheme with redundancy in coordinate mapping of graph for CAM-based scheme highlighted in yellow.

bandwidth inside RRAM to execute matrix multiply-accumulation (MAC) in-memory, mitigating the data movement along with high cost. However, we observe two major inefficiency problems in the design of RRAM-based SpMV accelerator. First, the sparse nature of SpMV operations forces the RRAM crossbar cells to store zero-edge's data as shown in Figure 1c, leading to ineffective and redundant computations in the MAC. This results in the increase of total number of MAC in each computing MAC crossbar. Second, these solutions still suffer from the irregularity, sparsity and redundancy properties that are common in typical graphs.

To address these issues, GaaS-X [5] adopts the RRAM-based content addressable memory (CAM) to search the non-zero edges and subsequently activate target edge's properties to participate in MAC operation. CAM-based array facilitates the parallel search of required vertex ID across all rows resident intra- and inter-crossbar, achieving precise computation exactly and avoiding performing the invalid zero edge located in padding data layout. With this method, GaaS-X stores the same destination vertex IDs of multiple different edges for many times (i.e., destination A, B, C and D are all stored twice in Figure 1d in yellow), incurring storage redundancy. As the number of edges grows, GaaS-X's inevitable redundancy of storage required as shown in Figure 1d (in yellow) leads to a significant amount of CAM array area.

Our analysis shows that the performance of RRAM-based PIM accelerator is bounded by the inefficiencies during the loading step, when the input data is being loaded into the accelerator. To improve efficiency, we first analyze the latency incurred by different operations in CAM-based scheme. CAM-based scheme (as shown in Figure 1d) executes five stages in graph running, including search, hint read, data indexing, MAC and update. The former three stages (search, hint read and data indexing) make up the data load in CAM-based scheme. In search stage, CAM array takes an activated

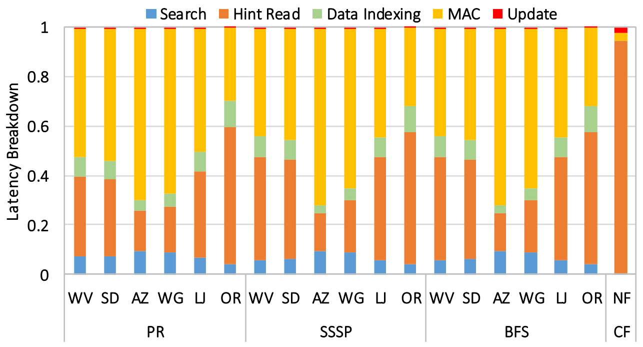

Figure 2: Latency breakdown in CAM-based scheme.

vertex ID to search the rows having its edges and output match signal. In hint read stage, the neighbor ID of activated vertex needs to be read from the CAM according the match for next property gather (For simplicity, we refer to the neighbor ID as *hint*, same below). In data indexing, the hint is fed into on-chip buffer to gather per-neighbor property. In MAC stage, the MAC crossbar performs the MAC operation to yield the new property of activated vertex. In update stage, the new property is updated into on-chip buffer. In Figure 2 of our evaluation in different datasets, the data load consisting of the search, hint read and data indexing consumes 51% time in average. Therefore, we get the observation that the CAM-based performance is bounded by inefficiency of data load.

Our design (Gzippo) is a highly-compact processing-in-memory accelerator for graph application based on the observation. To alleviate inefficiency of data load, Gzippo fully combines the parallel property of classical crossbar and the compact attribute of CSR format to fulfill a tandem-isomorphic-crossbar architecture. Gzippo's multi-banked buffer and simple-designed crossbar avoid serialization of reading input data. Gzippo decouples the data search and the hint read stage by designing an additional logic for computing the location of required data. With the help of the specific logic, Gzippo eliminates the CSR format's serialization overhead while benefiting from a much more compact format, eliminating the redundancy of CAM-based schemes. Our evaluation shows that Gzippo achieves 3.6× ~70.0× performance improvement over SpMV-supported scheme, 1.6×~17.4× over CAM-based scheme, and 72.0×~3916.0× energy efficiency improvement over SpMV-supported scheme, 13.1×~163.2× energy efficiency improvement over CAM-based scheme.

Our contributions can be presented as follows: (1) We design Gzippo, a RRAM-based tandem-isomorphic-crossbar accelerator architecture consisting of hint indexing crossbar and MAC computing crossbar for general graph applications, adapting the highly-compressed CSR format to reduce the area and overall cost. (2) We provide a RRAM-based parallel prefetch scheme on graph, enabling a high-throughput multi-banked buffer for MAC crossbar. (3) We construct a dedicated PIM-inside pipeline to optimize the execution latency based on abstract properties exhibited by general graph algorithms.

2 BACKGROUND & MOTIVATION

2.1 RRAM-enabled Processing

Resistive random access memory is known for the non-volatility, still enabling the analog computation in situ mode rapidly. As shown in Figure 3a, RRAM-based crossbar array can perform vector-matrix multiplication (VMM) operation by supporting multiple in-memory vector-vector multiplications in parallel. Before executing VMM, the element of matrix is mapped into the resistance in each crossbar cell physically and the input vector is represented as the voltage signal from the digital to analog converter (DAC). This input is then fed into the crossbar array, which aggregates the current from each column by Kirchhoff's law. Finally, the sample and hold (S&H) extracts and pipes the column current into analog to digital converter (ADC), yielding the digital resulting vector. Through this process, a vector-matrix multiplication can be done with no additional data movement between the processing core and memory.

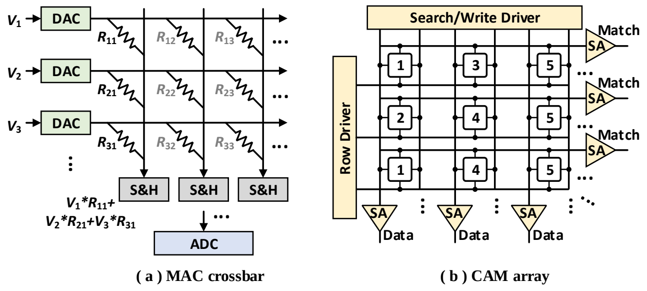

Figure 3: RRAM-enabled VMM and search operation.

2.2 Inefficiency in RRAM Graph Accelerators

Shared Resources. While these accelerators provide high performance, we find that the current graph representation required leads to inefficiency. RRAM-based CAM array consisting of complement memory elements [15] powers selective search operation in a large crossbar, shown in Figure 3b, vertex ID for 1, 2, 3, 4 and 5 stored in a 3×3 CAM. Vertex 1 and 2 in same column (or 3 and 4 in same column) can be check with target vertex for whether matched or not concurrently. If there are one of two matches, the corresponding horizontal match line will output a match signal to sense amplifier (SA). Notably, vertex 1 and vertex 2 cannot be forwarded to the output at the same time and need to be acquired sequentially through vertical bit line. This situation is similar in vertex 3 and vertex 4. When the CAM enables the parallel search in graph analysis, most of graph algorithms require many iterations for execution, e.g., PR. Hence, previous iterated intermediate rank value from PR application needs to be sent in the computing MAC crossbar. After obtaining the information of rows necessary for joining into MAC, the last rank values of each source vertices from same destination vertex need to be fetched from buffer and fed into the input registers of computing crossbar, as shown in Figure 1d (in gray). GaaS-X should sequentially read the source vertex ID for indexing intermediate iterated value in on-chip buffer, exacerbating latency of load to impact computing performance of MAC crossbar (shown in Figure 2).

Graph Representation. The irregularity and flexibility expressed in graph render specific format to represent this type of data structure effectively. Array representation of graph stores large volume of invalid zeros, suffering from serious sparsity. Coordinate format (COO) only keeps the valid edge, excluding the sparsity from the zero edges as shown in Figure 1b, where COO stores each edge in form of (source vertex, destination vertex, weight) from graph

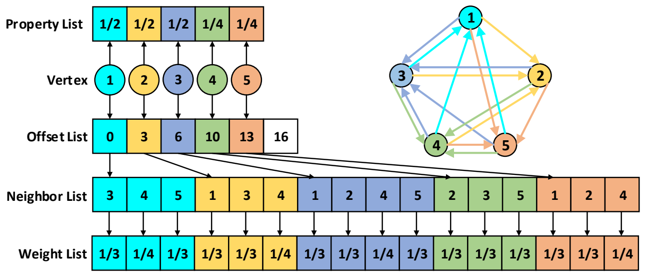

Figure 4: Graph sample stored as CSR format.

sample in Figure 1a, e.g., (C, A, ✓) represents the edge from vertex C to vertex A with a weight as ✓. But redundancy from the overlapped storage of common vertex located in several edges can lead to large memory footprint. Instead, the compressed sparse row (CSR) format enjoys the space efficiency on graph representation without sparsity and redundancy.

Instead of the COO format, the CSR format consists of four lists as shown in Figure 4, such as offset list, neighbor list, weight list and property list. The neighbor vertices of each vertex are put together in neighbor list closely and continuously. Each vertex has a offset value, designating the scope of its edges for access. For weighted graph, the weight of each edge stands in weight list in line corresponding to its related neighbor. Vertex property list takes the properties of all vertices in increasing order. Figure 4, as a directed graph, is stored as pull mode. Taking the vertex 2 as an example in Figure 4, all values in yellow reflect the information related with vertex 2. Vertex 2 has a property as 1/2 in the property list. The offset value of vertex 2 (i.e., 3 in offset list) represents its edges stored in neighbor list starting from the 3-th location to next vertex offset (i.e., 6 of vertex 3). Vertex 2 just records its 3 incoming edges in pull mode, such as (1, 2), (3, 2) and (4, 2). (e.g., (1, 2) represents a directed edge from source vertex 1 to destination vertex 2.) Correspondingly, each edge of vertex 2 has a weight in the weight list, such as 1/3, 1/3 and 1/4 (e.g., 1/3 is the weight of edge (1, 2)).

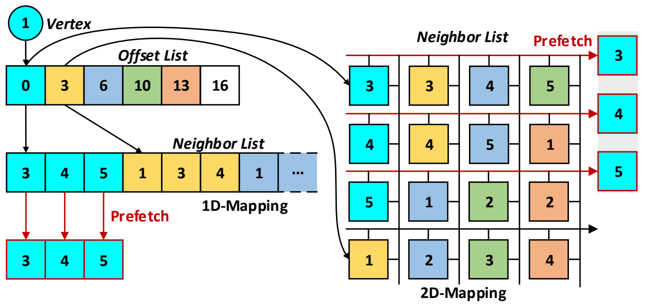

Figure 5: Graph data layout in CSR format and prefetch mechanism

Since the CSR format includes an offset list of all neighbors of each vertex and enjoys high compressed degree removing spatial inefficiency (shown in Figure 1c and Figure 1d), we can exploit these properties to prefetch multiple neighbors concurrently in a timely fashion. Before the computation, the graph algorithm needs to access the neighbor list. As shown in Figure 5, the 1D mapping contains the locations of all neighbors of each vertex and puts them together in line. Therefore, the 0 and 3 from offset list indicate that the first 3 neighbors (i.e., 3, 4 and 5) are the neighbors of vertex 1. In this case, we can exploit this information to prefetch data in the 2D mapping within the RRAM's regular crossbar in column-major way and read the offset (0, 3). Then, the accelerator should do the remainder and division operations on the offset (0, 3) to get the locations in first column and first three rows. In our design, we leverage this CSR format's highly-compressed scheme to organize neighbor information and allowing more data to fit within the limited RRAM crossbar. Neighbors of a given vertex can be prefetched with the help of hints indirectly provided by offset. The idea behind our Exactly Hint Indexing (in Section 3.2) is that CSR format provides offset for easily-and-exactly indexing of data onto the regular RRAM crossbars through simple remainder and division operation.

3 GZIPPO DESIGN

Unlike the COO format where CAM supports the serial search for data gathering, the CSR format requires simple RRAM cell to assist in gathering all necessary data first before the computation. To this end, we propose two additional components: *tandem-isomorphic crossbar* and *specific location identifier logic* to Gzippo. Our design allows parallel prefetch-and-load capacity to match the rapid analog computing speed of RRAM PIM architecture, leading to a high-performance RRAM graph accelerator at the low cost.

3.1 Gzippo Microarchitecture

We first discuss the components and the high-level design of Gzippo. Figure 6 shows the tandem-isomorphic crossbar of Gzippo. Gzippo exploits both storage and computation capabilities of the RRAM crossbar to formulate two different types of crossbars: hint crossbar (❹) and MAC crossbar (❻). Hint crossbar only takes advantages of the storage functionality of RRAM. MAC crossbar possesses both storage and computation functionalities. Multiple hint crossbars and MAC crossbars constitute hint matrix and MAC matrix, respectively. In Gzippo, we store the neighbor list in the hint matrix and the weight list in the MAC matrix (as shown in Figure 4). To enable parallel prefetch, Gzippo exploits the common RRAM cell for the hint crossbar cell and MAC crossbar cell without additional design complexity. The MAC crossbar and the hint crossbar shares most components except for the representation of bit and some peripheral logic. For MAC crossbar, one cell represents two bits, but for hint crossbar, one cell represents one bit. For peripheral circuitry, hint crossbar just needs SA to support read while MAC crossbar includes additional S&H to fulfill the analog computation of MAC.

To boost tandem-isomorphic crossbar performance, Gzippo includes on-chip buffers and specific logic to enable parallel prefetch-and-load capacity. This parallel prefetch-and-load are designed to allow prefetching of immediate data being iterated over multiple graph vertices and transfer these data input into MAC matrix for the next update. This can be done through three components. First, to find all edge information of each vertex in the CSR format, Gzippo includes the offset buffer (❷) as additional buffer that can take the offset of each vertex. Second, to accurately prefetch immediate

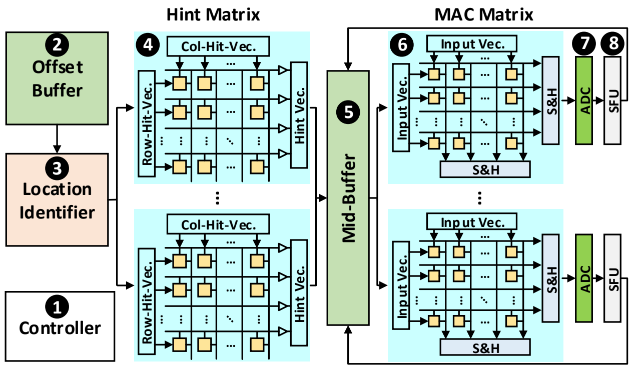

Figure 6: Gzippo architecture overview.

values, Gzippo introduces the location identifier (❸). The location identifier is used as an ad-hoc logic unit that receives the offset tuple from offset buffer and produces the row-/column-hit-vector for hint matrix (discussed in Section 3.2). Third, all hint data is then forwarded to an additional on-chip buffer, called the *mid-buffer* (❺), which stores the prefetched data and orchestrates the input data for the MAC matrix. The special function unit (SFU) (❽) performs shift-and-add operation in order to sum multiple column resulting values inside the MAC crossbar.

To schedule all the components, Gzippo maintains a central controller (❶) that orchestrates the neighbor list on hint matrix and edge properties or weights into MAC matrix before launching computation. During processing progress, the central controller also needs to be able to coordinate all the operations required in execution. In summary, Gzippo's main structural components include the hint matrix, the MAC matrix, the on-chip offset buffer, the mid-buffer, the location identifier, the central controller and the SFU.

3.2 Exact Hint Indexing

The primary goal of the Exact Hint Indexing is to ensure that the graph data in the CSR format is aggregated and provide Gzippo's computation unit with correct graph information. To allow accurate data aggregation for a graph operation in CSR format, Gzippo uses the row-hit-vector to get the target's vertex IDs from the hint matrix exactly. To do this, we first communicate with the software runtime of Gzippo specification (i.e., the number of rows, columns as well as the total number of the crossbars in our Gzippo structure). Then, the offset buffer is populated with the offset values that are required for the current iteration of the MAC operation. This offset buffer is then used by the location identifier to determine the actual address to be prefetched into the MAC matrix (shown in Figure 7a). Using the data from Figure 4 as an example for PR application, the offset tuple points out the starting and ending store position of edges belonged to a vertex in the offset list, e.g., (0, 3) is the offset tuple of vertex 1. With this offset tuple information, our location identifier performs the remainder and division operation on the offset tuple using Gzippo's structure size (i.e., (1, 4, 4) refers to one 4-by-4 crossbar), which yields the crossbar-column-row location span resulting in the tuple: {*starting location, ending location*}. The starting and ending location tuple are tagged with the crossbar

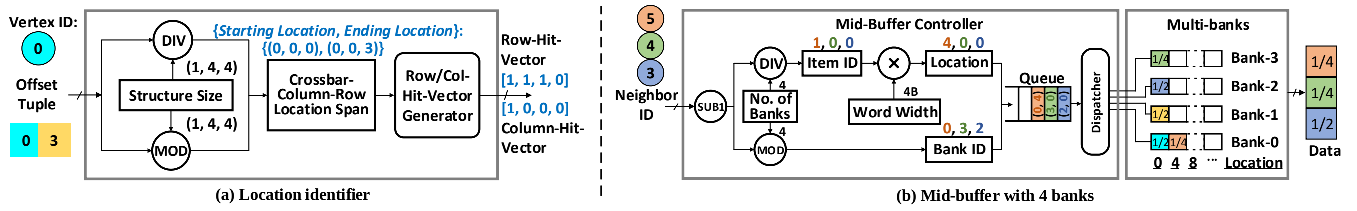

Figure 7: Microarchitecture of the location identifier and mid-buffer.

label, column label and row label to formulate the location triple (using the earlier example from Figure 4, the ending location triple of vertex 1, with the crossbar label 0, is (0, 0, 3)). Since the CSR format puts all neighbors of one vertex together and the hint/MAC matrices adopt the column-major way to map neighbor and weight list into crossbars consecutively, the crossbar-column-row location span {(0, 0, 0), (0, 0, 3)} indicates that the neighbor of vertex 1 stand continually in the RRAM from first-crossbar-first-column-first-row cell to first-crossbar-first-column-second-row cell. The last step of the exact hint indexing is to generate the next set of vectors. To fulfill this, the row-/column-hit-vector generator receives the crossbar-column-row location span and *produce the row-hit-vector and column-hit-vector* (e.g., row-hit-vector [1, 1, 1, 0] and column-hit-vector [1, 0, 0, 0]) for the next search and computation. This vector is used as a filter for the content to be fetched. With the help of the hit-vector, the hint matrix and the MAC matrix utilize the indexing information to gather data and perform the computation. In the case that the edges of a vertex are populated in several different columns or crossbars, the location identifier needs to consider the bound information and offers multiple hit-vectors for downstream crossbars.

3.3 Parallel Data Stream

Graph information can be prefetched into the execution section of Gzippo's compute components. To perform this task in parallel, we add the on-chip mid-buffer as shown in Figure 7b. The mid-buffer exploits a multi-banks scheme to store the intermediate value on chip in round-robin bank-interleaving way, e.g., the intermediate value of vertex 1~4 is stored in the first location of bank 0~3 in order and the intermediate value of vertex 5 is put into the second location of bank 0 in round-robin. This type of mapping can contribute to reducing the bank conflict and improving the read parallelism. With this structure, the central controller can feed the row/column-hit-vector into hint matrix and acquire *multiple hints* in parallel, e.g., 3, 4 and 5 in Figure 7b. These hints are then stored in mid-buffer to index the previous property values of vertices. Mid-buffer controller calculates the location of the data in mid-buffer (i.e., bank ID (0, 3, 2) and location (4, 0, 0) in bank) for each hint and multiple locations are then put into the queue. Next, the dispatcher issues several locations' information from the queue into multiple banks. The multi-banks output several intermediate values into the input register of MAC matrix. Here, the bank-0, 2 and 3 in Figure 7b are activated and output three property data at one time. This enables one key benefit: the dedicated mid-buffer enables the multiple intermediate values to be fed into MAC matrix at same time and avoids serialization when intermediate values are being used.

With the offset information available in the CSR format, our Gzippo design can utilize the hint indexing structure to further prefetch the location information of the next iteration from hint matrix into the mid-buffer using the offset located in CSR format. The mid-buffer structure allows our design to decouple the execution flow from the dedicated PIM+CSR-based data fetching pipeline.

3.4 Operating Graph Applications

We use the page rank algorithm as an example to demonstrate how Gzippo operates. As shown in Figure 8, the controller first initializes Gzippo with input data from Figure 4. It stores the serialized neighbor list onto the hint crossbar column by column. The weight list is stored into the MAC crossbar in same way. The controller fetches offset list into offset buffer. Once firing the iterating flow of PR algorithm, an offset tuple of a given destination vertex 1 is read from the offset list resided in offset buffer (ⓐ) and then transmitted into the location identifier for yielding row-/column-hit-vector (ⓑ). The row-hit-vector designates which rows in both hint and MAC crossbar need activating for current destination vertex update (row 0, 1, and 2), same as the column-hit-vector for column activation (0-th column). When activated by input signal, the hint crossbar outputs the vertex ID 3, 4, and 5 concurrently into the output register and then into mid-buffer (ⓒ) for searching the previous rank value. Deserved being notably, Gzippo architecture can obtain ID-3, 4, and 5 at one cycle, that is, no matter how many edges of some destination vertex are located in one columns, hint crossbar can concurrently output these vertex IDs into the mid-buffer for indexing intermediate rank value into MAC crossbar (ⓓ). Besides, one vertex typically has more neighbors, and will be accelerated via parallel loading. Note that this is different from CAM-based scheme because CAM array cannot output several vertex IDs into on-chip buffer for indexing intermediate value, through sequentially outputting vertex ID one-by-one (as discussed in Section 2.2). Gzippo can yield better performance speedup from parallel loading. After all intermediate values are prepared as an input vector, the MAC crossbar will execute the MAC operation (ⓔ) in higher-speed analog way. The updated rank result (ⓕ) from selective corresponding column will override the old one if meeting the constrains. Gzippo iterates other destination vertices one-by-one until all iterations converge. The whole computing flow can be optimized in a pipelined manner to further improve performance.

4 EVALUATIONS

We now provide the experimental methodology, graph benchmark, dataset and evaluation results compared with the state-of-the-art baselines.

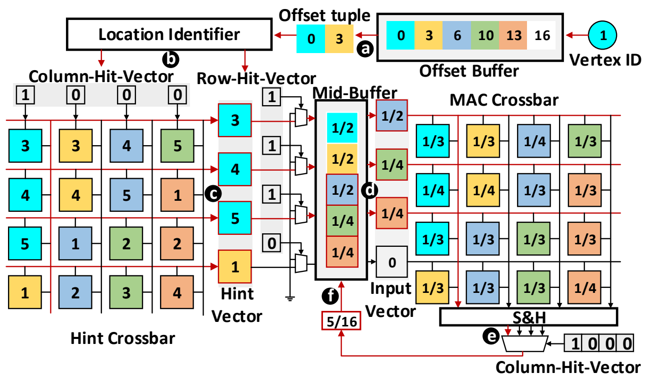

Figure 8: Gzippo executing mechanism for PR.

4.1 Experiment Setup

System Design and Evaluation. For the performance and energy cost evaluation, we build a dedicated in-house cycle-level simulator to implement the Gzippo architecture design. The simulator establishes and takes the key characteristics of RRAM-based PIM architecture: peripheral circuit, on-chip buffer (including offset buffer, mid-buffer, input and output buffer) and additional function unit into account. We use the CACTI-7 [2] to model the on-chip buffer at 32 *nm* technology node (same as GaaS-X) to acquire the latency and power parameter. Moreover, we get the read/write latency and read/write energy cost of PIM compute component from [21], 29.31*ns*/50.88*ns*, 1.08*pJ*/2.91*nJ*, respectively. The custom simulator can achieve the system-level execution time and energy cost profiling.

Table 1: Parameters of our Gzippo Architecture.

Component	Configuration	Area ($mm^2 \times 10^{-3}$)	Power (mW)
MAC Crossbar	128×128 2-bit/cell number: 2048	51.20	307.20
Hint Crossbar	128×128 1-bit/cell number: 1024	40.00	310.00
ADC	6-bit, 1.2 *GSps* number: 512	300.80	328.96
DAC	2-bit/cell number: 256×2048	0.08	1.64
S&H	number: 256×2048	20.48	5.12
Digital Circuitry	I.Location Identifier II.Controller III.SFU	1524.90	71.44
Input Buffer	16 KB	6.40	8.72
Output Buffer	64 KB	25.60	34.88
Offset Buffer	512 KB	204.80	279.04
Mid-Buffer	512 KB	204.80	279.04
Total		2.38 mm^2	1.63 W

We implement the digital control circuitry including location identifier, controller, multiplexers and SFU units in System Verilog RTL using the 32 *nm* (same as GaaS-X) technology node [27]. For operation at 1 *GHz*, we get the latency and power cost parameters of digital control circuitry. We configure CAM's parameter based on GaaS-X specifications [5]. Because Gzippo can activate up to 16 rows in MAC crossbar through the row hit vector from hint crossbar, we utilize a 6-bit ADC to support the accumulation in each bitline. We use the DAC and ADC model based on the design from [12]. We use NVSim [9] to evaluate the area, latency and power consumption of RRAM crossbar. The overall area and power consumption of the Gzippo design is 2.38 mm^2 and 1.63 W, respectively. Table 1 lists all the parameters of Gzippo.

Dataset and Baseline Table 2 summarizes all the datasets used in our evaluation. The WikiVote (WV) [14], SlashDot (SD) [14], Web-Google (WG) [14], Amazon (AZ) [14], Orkut (OR) [14], LiveJournal (LJ) [14] can evaluate the PageRank, SSSP and BFS algorithms. The remaining Netflix [4] dataset is used to test the CF and we set the feature size as 32. We mainly compare Gzippo with state-of-the-art RRAM-base PIM accelerators. One is SpMV-supported GraphR [25], the other is CAM-based GaaS-X [5]. We model the characteristics of GaaS-X employing our cycle-level simulator with same hardware parameters same as Gzippo. Notably, compared with GaaS-X with 2048 CAM crossbars, Gzippo only uses 1024 hint crossbars to facilitate execution.

Table 2: Graph Dataset and Descriptions.

Datasets	No. of Vertices	No. of Edges
WikiVote Data (WV)	7.0K	103K
SlashDot Network (SD)	82K	948K
Webgraph in Google (WG)	0.88M	5.1M
Amazon Network (AZ)	262K	1.2M
Orkut Network (OR)	3.0M	106M
LiveJournal Network (LJ)	4.8M	69M
Netflix Data (NF)	497.8K	99M

4.2 Performance

Figure 9 shows the speedup in performance of Gzippo with respect to GraphR [25] and GaaS-X [5] on diverse graph datasets for PR, SSSP and BFS algorithm. Compared with GraphR and GaaS-X, Gzippo achieves 19.5× and 3.0× speedup on average. We observe the prefetch scheme is able to find the target data at the timely manner, allowing the data to be pipelined into the crossbar architecture in parallel, which leads to the performance improvement over the state-of-the-art baselines. We also observe the significant performance improvement of 17.4× over the GaaS-X baseline in CF algorithm over GaaS-X due to exactly hint indexing on large amount of neighbors, which leads to a significant overhead in GaaS-X.

4.3 Energy Efficiency

As show in Figure 10, Gzippo is more energy-efficient than the GraphR and GaaS-X in different datasets for all four types of graph algorithms. Gzippo acquires a 495.2× and 23.9× in geometric mean speedup over GraphR and GaaS-X, respectively. While GraphR spends much energy in large volume of invalid zero-edges computations, GaaS-X consumes much energy in searching the target vertex among all crossbars at each update. In contrast, we observe that Gzippo only has to activate and search on the necessary crossbars with the help of our prefetching mechanism. This allows Gzippo to avoid extra energy needed to search and perform data lookup over multiple iterations. This is especially beneficial when a vertex shares more neighbors such as the NF dataset, where Gzippo achieves a thousands- and hundreds-level energy efficiency compared with GraphR and GaaS-X, respectively.

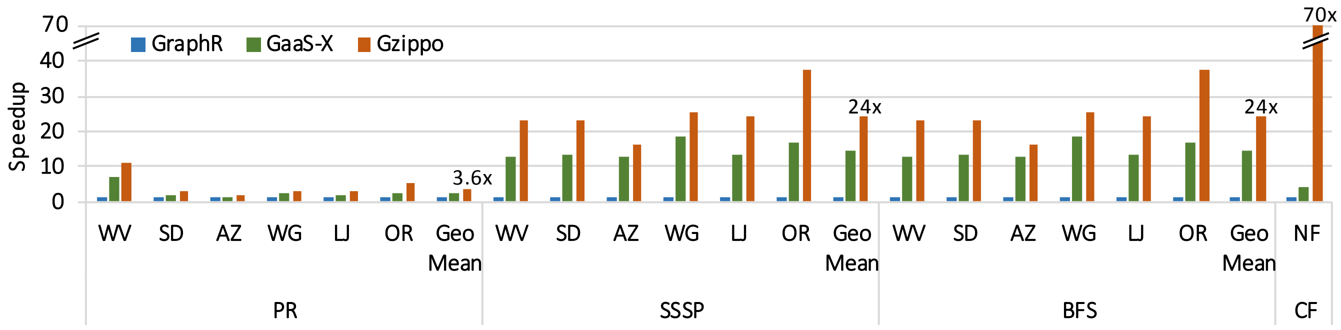

Figure 9: Performance normalized to GraphR.

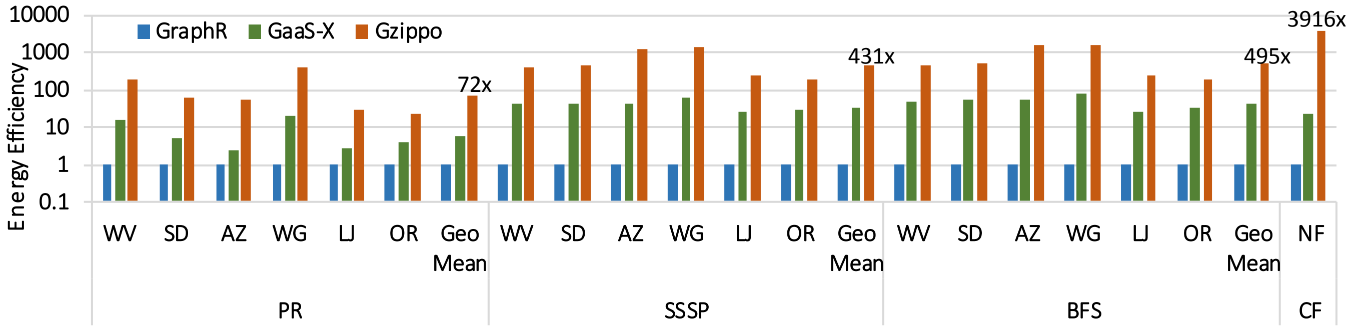

Figure 10: Energy efficiency normalized to GraphR.

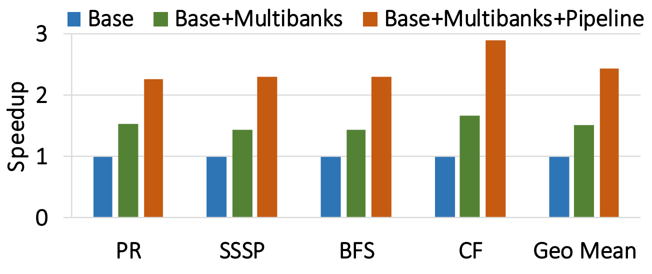

Figure 11: Ablation results for different optimizations.

4.4 Ablation and Sensitivity Study

Figure 11 shows performance contribution from different parts designed in Gzippo. We increasingly introduce design to demonstrate the effectiveness of designs. The *base* indicates the based idea of Gzippo without multibanks and pipeline. The *base+multibanks* means only to introduce the multibanks. The *base+multibanks+pipeline* refers to optimize the Gzippo in base+multibanks in pipelined manner. We chose the base mode as the baseline, other modes normalized to the base. Figure 11 shows that both multibanks and pipeline can contribute to the performance improvement compared with the base.

Figure 12 shows the sensitivity study of the crossbar size with the SSSP algorithm in SD dataset. We evaluate with the crossbar size ranges from 64×64 to 2048×2048. We observe that Gzippo remains effective at diverse crossbar size, length and width of crossbar. With the regard to the energy efficiency, the line curve in Figure 12 shows that energy efficiency of Gzippo over GaaS-X reduces gradually and then plateau as the size increases. Since the crossbar becomes larger and can store more data in same crossbar, the GaaS-X can fetch the target vertex using fewer crossbars and thus wastes less energy in search, equal to Gzippo in extremely large crossbar.

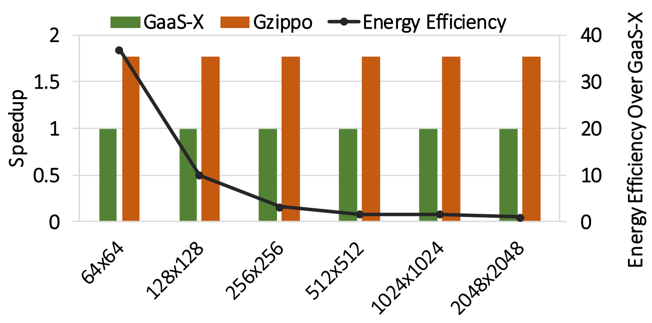

Figure 12: Sensitivity result based on different size of crossbar (Bars) The speedup of Gzippo with GaaS-X as baseline (Line) The energy efficiency of Gzippo over GaaS-X.

5 RELATED WORK

Hardware-based Graph Accelerators. Graph-specific memory hierarchy design is attracting more attention as data can be orchestrated so they are readily available to the CPU at the cache. To exploit the locality of hot vertices, GRASP [10] is proposed to maximize the reuse of hot vertices while Graphicionado [13] is designed to cache highly-reused graph data using on-chip scratchpad. Meanwhile, Mukkara et al. propose HATS [20], which utilizes an online scheduler to traverse the locality inherent in cluster property of the real-world data. Basak et al. [3] propose DROPLET with a data-aware prefetcher and alleviate the serialization from dependency chains of different types of graph data. While these works can accelerate graph application, they do not take advantage of the available compute capability of new technologies. Gzippo directly exploits properties of both RRAM and CSR format to accelerate graph computation in RRAM..

RRAM-based In-memory Accelerators. Processing-in-memory provides a novel type of paradigm for addressing the memory wall existed in traditional accelerators. Tesseract [1] introduces the dedicated processing unit near DRAM to employ the large internal bandwidth for graph analytics acceleration. Unlike Tesseract, GraphR [25] utilizes the analog MAC capacity of RRAM and largely parallel computation of RRAM crossbar to enable the SpMV, but with sparsity. GRAM [31] leverages digital RRAM-based technique and supports the compare-and-swap (CAS) when improving the parallelism available in the graph computation through bit-wise operations. Different from GRAM and GraphR, GaaS-X [5] uses the CAM-based search and MAC analog compute to increase the performance of graph analytics, along with redundancy in RRAM. Notably, Gzippo introduces the tandem-isomorphic-crossbar design and exactly hint indexing mechanism to reduce both sparsity and redundancy in graph data, leading the improvement in performance.

Not only in graph domain, many previous works utilize RRAM for deep learning applications. PRIME [6] configures RRAM crossbar as neural network accelerator with substantial performance. ISAAC [23] firstly proposes a deep pipeline into RRAM-based architecture to increase the the throughput for convolutional neural networks (CNN). PipeLayer [24] exploits inter-layer parallelism to boost the performance of both training and inference for CNN. FORMS [29] employs a fragment polarization method to reduce the RRAM crossbar cost for deep neural networks, yet ameliorating the performance.

Software Approaches to Graph Accelerations. Rather than hardware accelerators, many prior works have explored graph accelerations through system software modifications and code optimizations. TurboGraph [16], GridGraph [32], GraphChi [17], and X-Stream [22] are CPU-based software approaches to graph accelerations. Similar to the hardware-based memory hierarchy design, these software approaches mainly focus on locality and data access. This can be done in software by partitioning the whole graph into shards and processing in parallel. Unlike GraphR [25], the GraphMat [26] abstracts the graph operations into SpMV computation in software manner with regard to CPU. Apart from CPU-related framework, Gunrock [28], nvGraph [8], MapGraph [11] and Enterprise [19] use the larger memory bandwidth of GPU than CPU to improve the performance. Though these frameworks are flexible and efficient on common servers, they fail to address the memory wall challenge from traditional memory hierarchy design. Gzippo addresses the challenge by performing the acceleration in memory and limits memory-to-core data movements.

6 CONCLUSION

We propose Gzippo, a highly-compact RRAM-based PIM accelerator still with high performance. Gzippo addresses *both* sparsity and redundancy typically happen in state-of-the-art RRAM graph accelerators. Gzippo fully exploits the data parallelism of RRAM-basd crossbar to enable the parallel data load into MAC crossbar. Gzippo also performs data prefetching to further improve performance. Gzippo achieves a 19.5× and 3.0× (up to 17.4×) performance speedup, 495.2× and 23.9× (up to 163.1×) energy efficiency over SpMV-supported and CAM-based scheme, respectively.

ACKNOWLEDGMENTS

We thank all anonymous reviewers. Thanks for the support from Biren Research. This work is partly supported by the National Key R&D Program of China (Grant No. 2018YFA0701500) and National Natural Science Foundation of China (Grant No. 61972242). The corresponding author is Xiaoyao Liang.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 105–117.

[2] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[3] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and optimization of the memory hierarchy for graph processing workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 373–386.

[4] James Bennett et al. 2007. The netflix prize. In *Proceedings of KDD cup and workshop*.

[5] Nagadastagiri Challapalle, Sahithi Rampalli, Linghao Song, Nandhini Chandramoorthy, Karthik Swaminathan, John Sampson, Yiran Chen, and Vijaykrishnan Narayanan. 2020. Gaas-X: Graph analytics accelerator supporting sparse data representation using crossbar architectures. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 433–445.

[6] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. 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)*. IEEE, 27–39.

[7] Thomas H. Cormen et al. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press.

[8] Nvidia Corporation. 2016. The NVIDIA Graph Analytics library (nvGRAPH). In *https://developer.nvidia.com/nvgraph*.

[9] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P Jouppi. 2012. NVsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 7 (2012), 994–1007.

[10] Priyank Faldu, Jeff Diamond, and Boris Grot. 2020. Domain-specialized cache management for graph analytics. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 234–248.

[11] Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of workshop on GRAph data management experiences and systems*. 1–6.

[12] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-memory data parallel processor. *ACM SIGPLAN Notices* 53, 2 (2018), 1–14.

[13] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[14] Leskovec J and Krevl A. 2014. SNAP Datasets: Stanford large network dataset collection. *Ann Arbor, MI, USA.* http://snap.stanford.edu/data

[15] Nicholas Jao et al. 2019. Programmable Non-Volatile Memory Design Featuring Reconfigurable In-Memory Operations. In *ISCAS*.

[16] Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++ A scalable and fast graph analytics system. In *Proceedings of the 2018 international conference on management of data*. 395–410.

[17] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 31–46.

[18] Page Lawrence et al. 1999. The PageRank citation ranking: Bringing order to the web. In *Stanford InfoLab*.

[19] Hang Liu and H Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[20] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–14.

[21] Dimin Niu, Cong Xu, Naveen Muralimanohar, Norman P Jouppi, and Yuan Xie. 2013. Design of cross-point metal-oxide ReRAM emphasizing reliability and cost. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 17–23.

[22] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[23] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. (2016), 14–26.

[24] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 541–552.

[25] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 531–543.

[26] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: high performance graph analytics made productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225.

[27] Synopsys. 2020. Teaching Resources for IC Design. https://www.synopsys.com/community/university-program/teaching-resources.html.

[28] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. 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*. 1–12.

[29] Geng Yuan, Payman Behnam, Zhengang Li, Ali Shafiee, Sheng Lin, Xiaolong Ma, Hang Liu, Xuehai Qian, Mahdi Nazm Bojnordi, Yanzhi Wang, et al. 2021. FORMS: fine-grained polarized ReRAM-based in-situ computation for mixed-signal DNN accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 265–278.

[30] F. Benjamin Zhan et al. 1998. Shortest Path Algorithms: An Evaluation Using Real Road Networks. *Transportation Science* (1998).

[31] Minxuan Zhou, Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. GRAM: graph processing in a reram-based computational memory. In *IEEE Asia and South Pacific Design Automation Conference*.

[32] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.