GraphRC: Accelerating Graph Processing on Dual-addressing Memory with Vertex Merging

Wei Cheng
p76091226@gs.ncku.edu.tw
Dept of Computer Science and Information Engineering, National Cheng Kung University
Tainan, Taiwan (R.O.C)

Chun-Feng Wu
cfwu417@cs.nycu.edu.tw
Dept of Computer Science, National Yang Ming Chiao Tung University
Hsinchu, Taiwan (R.O.C)

Yuan-Hao Chang
johnson@iis.sinica.edu.tw
Institute of Information Science, Academia Sinica
Taipei, Taiwan (R.O.C)

Ing-Chao Lin
iclin@mail.ncku.edu.tw
Dept of Computer Science and Information Engineering, National Cheng Kung University
Tainan, Taiwan (R.O.C)

ABSTRACT

Architectural innovation in graph accelerators attracts research attention due to foreseeable inflation in data sizes and the irregular memory access pattern of graph algorithms. Conventional graph accelerators ignore the potential of Non-Volatile Memory (NVM) crossbar as a dual-addressing memory and treat it as a traditional single-addressing memory with higher density and better energy efficiency. In this work, we present GraphRC, a graph accelerator that leverages the power of dual-addressing memory by mapping in-edge/out-edge requests to column/row-oriented memory accesses. Although the capability of dual-addressing memory greatly improves the performance of graph processing, some memory accesses still suffer from low-utilization issues. Therefore, we propose a vertex merging (VM) method that improves cache block utilization rate by merging memory requests from consecutive vertices. VM reduces the execution time of all 6 graph algorithms on all 4 datasets by 24.24% on average. We then identify the data dependency inherent in a graph limits the usage of VM, and its effectiveness is bounded by the percentage of mergeable vertices. To overcome this limitation, we propose an aggressive vertex merging (AVM) method that outperforms VM by ignoring the data dependency inherent in a graph. AVM significantly reduces the execution time of ranking-based algorithms on all 4 datasets while preserving the correct ranking of the top 20 vertices.

CCS CONCEPTS

• **Hardware → Non-volatile memory**; • **Software and its engineering** → *Main memory.*

KEYWORDS

graph processing, RC-NVM, dual-addressing memory, utilization, over-read, vertex merging

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549408

ACM Reference Format:
Wei Cheng, Chun-Feng Wu, Yuan-Hao Chang, and Ing-Chao Lin. 2022. GraphRC: Accelerating Graph Processing on Dual-addressing Memory with Vertex Merging. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3508352.3549408

1 INTRODUCTION

Graphs are widely used to express the relationship between data. To efficiently accommodate a large graph in memory devices for processing, Non-Volatile Memory (NVM) is one of the promising candidates to take the role of Dynamic Random-Access Memory (DRAM), which suffers from high unit costs and leakage power [14]. As NVM technologies become mature, a new NVM device, called Row-Column-NVM (RC-NVM) [17, 18, 24], is launched to integrate NVM technologies with the crossbar architecture. RC-NVM can not only provide larger and greener memory capacity than DRAM, but can also support a new memory access paradigm, that is the dual-addressing capability. In contrast to traditional memory devices which can only access in one direction (e.g., only-row or only-column), RC-NVM can access data either from the row or column direction. This dual-addressing capability can effectively improve the performance of graph processing. However, we found that the size mismatching between the graph data and the cache block[1] might reduce the cache block utilization rate while the system only accesses a few data in the cache block. The observed low cache block utilization issue of RC-NVM motivates this work to merge graph requests by selecting appropriate memory instructions (i.e., row-oriented or column-oriented). Merging graph requests eventually generates a higher cache block utilization rate and leads to better performance on RC-NVM.

The size of most real-world graphs is gigantic and thus it is expensive and energy-inefficient to store the whole graph in DRAM. Memory extension [21, 27] is a practical solution to meet this huge memory demand with lower costs. It works by storing the whole graph in storage devices and moving the required sub-graph to memory devices on-demand [30, 32]. However, intensive graph data movements between memory and storage devices is a critical issue for the memory extension solution. Fortunately, the manufacturing improvement makes storage and memory devices more powerful so the data movement costs between memory and storage devices are effectively reduced [23, 27]. For example, ultra-low-latency storage devices (e.g., NVDIMM [2, 26] and Samsung z-NAND Solid-State Drives (SSDs) [6]), which hit the market in 2017,

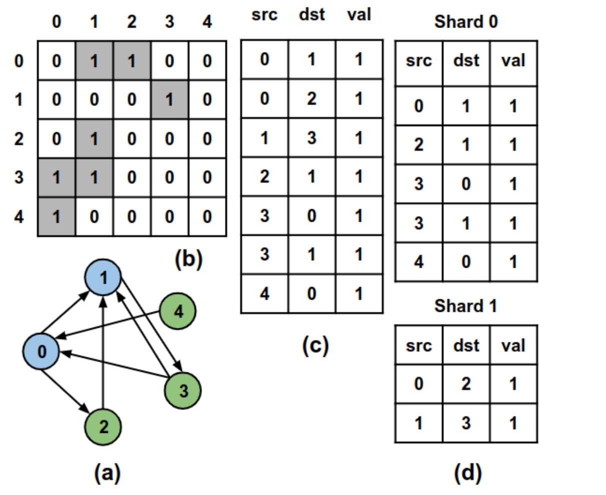
[1]The minimum access granularity of RC-NVM is a cache block, usually 64 bytes.

can provide ten times faster access latency than traditional SSDs. In addition to storage devices, NVM devices can provide near-DRAM access latency with lower unit costs and nearly no leakage power compared with DRAM. NVM devices (e.g., Phase Change Memory (PCM) and ReRAM) have drawn huge attention in several research fields, such as neural networks [8, 25, 28] and random forests [3, 9].

Mature manufacturing technologies not only enhance the device capability but also open up new prospects for the integration of memory devices and novel architectures. The ReRAM-based crossbar accelerator, which is one of the famous integration between ReRAM technologies and crossbar architectures, provides the capability to run matrix multiplication operations inside the device [7, 22]. Due to its distinctive specialization in matrix multiplications, the use case of the ReRAM-based crossbar accelerator is limited to applications that require intensive matrix multiplications, such as neural networks. Moreover, ReRAM-based crossbar architecture still faces several critical challenges which hinder this device from being production-ready. For instance, its peripheral circuits (e.g., the digital-to-analog converter and the analog-to-digital converter) suffer from high area costs [12, 31] and its sensing accuracy is unacceptable when more currents are summed on a bitline [1, 4, 10]. In addition to ReRAM-based crossbar accelerators, RC-NVM, which is a novel memory device also based on the integration between ReRAM technologies and crossbar architectures, supports dual-addressing capability with insignificant area costs [17]. With the support of dual-addressing capability, a memory location can be referenced in either row-oriented address space or column-oriented address space. In contrast to DRAM supporting only single-addressing memory, RC-NVM provides a new memory access paradigm which could accelerate the access performance of 2D data structures [24], such as tabular and graph structures widely used in databases and graph systems, respectively.

Although the capability of dual-addressing provides an opportunity to improve the performance of accessing graph data, some memory accesses suffer from low-utilization while deciding to use row-oriented or column-oriented accesses without considering the graph layout. Due to the size mismatching between the graph data and the cache block size, graph systems are required to issue extra memory requests when the utilization of some cache blocks is low. To efficiently access the graph data stored in RC-NVM, we propose GraphRC to improve the cache block utilization by selecting suitable memory instructions (i.e., row-oriented or column-oriented) for merging while considering the graph data layout on RC-NVM. We summarized our contributions as follows:

- We show that graph processing can be accelerated on dual-addressing memory by mapping in-edge/out-edge requests to column/row-oriented memory accesses respectively.
- We identify that some memory accesses still suffer from a low utilization rate on dual-addressing memory due to the size mismatching between graph data and the cache block size. Therefore, we propose a vertex merging (VM) method that improves the cache block utilization rate by merging memory requests from consecutive vertices. VM reduces the execution time of all 6 graph algorithms on all 4 datasets by 24.24% on average.
- We perform a detailed analysis on data dependency inherent in a graph and identify the factor that limits the effectiveness of VM is the percentage of mergeable vertices in a graph. Based on this observation, we propose aggressive vertex merging (AVM), a method that merges vertices based on the importance of a vertex, not its data dependency. AVM significantly reduces the execution time of ranking-based algorithms on all 4 datasets while preserving the correct ranking of the top 20 vertices.



(b)

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	0	1	0
2	0	1	0	0	0
3	1	1	0	0	0
4	1	0	0	0	0

(c)

src	dst	val
0	1	1
0	2	1
1	3	1
2	1	1
3	0	1
3	1	1
4	0	1

(d) Shard 0

src	dst	val
0	1	1
2	1	1
3	0	1
3	1	1
4	0	1

Shard 1

src	dst	val
0	2	1
1	3	1

Figure 1: A graph stored in (b) adjacency matrix format or (c) COO format or (d) shard format.

The rest of this paper is organized as follows. Section 2 presents the background, observation and motivation of this paper. Section 3 presents the proposed methods: VM and AVM that reduce the overall execution time of graph algorithms on dual-addressing memory. The experimental results are reported and discussed in Section 4. Section 5 concludes this paper.

2 BACKGROUND, OBSERVATION AND MOTIVATION

2.1 Graph Representation

Graphs are usually used to express the relationship between data. Each data is represented as a vertex and two related vertices will be connected by an edge, where the value of the edge represents how relative the two connected vertices is. Picked any vertex as a source in a directed graph, all edges pointing to this source are called in-edges and all edges pointing from this source to other destination vertices are called out-edges. For example, a directed graph is shown in Figure 1(a), where vertex 1 has three in-edges and one out-edge. To store a graph in computing systems, an adjacency matrix is a common way as shown in Figure 1(b), where row and column indices represent the source and destination vertices, respectively. Each element in matrix stores the corresponding edge value. However, adjacency matrices are usually sparse because graph data usually show power-law distribution [19], i.e., most vertices will point to very few hub vertices. To avoid this sparsity characteristic wasting too much memory, a coordinate list (COO) format is widely used to effectively store a graph in memory. As shown in Figure 1(c), a list comprises source-destination pairs, where each pair contains three fields: *src*, *dst*, and *val* representing an index of a source vertex, an index of a destination vertex and their corresponding edge value, respectively.

In contrast to an adjacency matrix, it is time-consuming to find a targeted source-destination pair in a COO list format because systems shall look up the list row-by-row in a brute-force way. To efficiently find a targeted pair in a COO list format, modern graph systems preprocess and convert a COO list format to a destination-based shard format. Practically, systems will first sort all source-destination pairs by their destination index (i.e., the *dst* field) and then partition the sorted COO list into several shards. Each shard consisted only of pairs with a certain destination indices, such as shard 0 in Figure 1(d) only contains pairs with their *dst* field setting to 0 or 1. To further improve the searching performance, all pairs in a shard are sorted by the source index.

Enterprises usually run link analysis techniques, e.g., PageRank and single-source shortest path, to extract useful information from the destination-based shard format. The conceptual model of the link analysis technique is that each vertex will first gather information from its neighbors by reading values on all in-edges, and then propagates the value by updating the results to all out-edges. For example, to run a link analysis on vertex 0 in Figure 1(a), it first accumulates the values on two in-edges associated with vertices 3 and 4, and then updates two out-edges associated with vertex 1 and 2. To read all values on in-edges, the graph system will read out the whole shard 0 in Figure 1(d) row-by-row from the memory and retrieve all values where their *dst* fields are 0. However, reading out all source-destination pairs from a shard is time-consuming and unnecessary. Obviously, a better way is to read out the *dst* column and then only read out two rows where their *dst* fields are 0. However, due to the constraint of the architecture of DRAM, data can only be read from one direction, which is row-oriented or column-oriented. This hardware constraint hurts the performance of graph processing for several decades.

2.2 RC-NVM Architecture

To unleash the constraint of traditional memory devices, a new memory device, called Row-Column-NVM (RC-NVM), was proposed in [17]. In contrast to traditional DRAM structures, RC-NVM employs non-volatile memory (e.g., ReRAM [11]) as its memory cells and adopts the crossbar architecture [5, 10]. Figure 2 shows the architecture of RC-NVM. The architecture of RC-NVM is similar to DRAM and is organized hierarchically as channel, rank, chip, bank, subarray, and Mat. RC-NVM enables dual-addressing ability by adding extra elements at different hardware levels. Both row buffer and column buffer are shared among banks in RC-NVM while there is only a stand-alone row buffer for each bank in DRAM. Furthermore, both write driver (WD) and sense amplifier (SA) are placed on both sides of a RC-NVM Mat such that a memory cell can be driven or sensed from either row or column direction. Thanks to this hardware characteristic, RC-NVM can read data either from row-oriented address space or column-oriented address space.

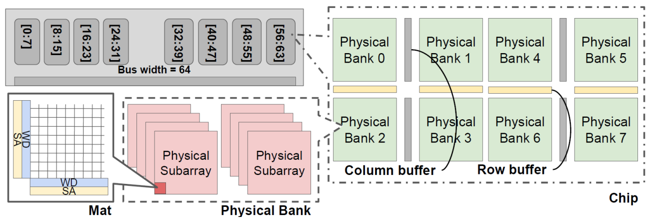


Figure 2: RC-NVM architecture

A logical subarray is a set of physical subarrays scatter over different chips within the same rank. As shown in Figure 3, eight physical subarrays among eight different chips are viewed as a single logical subarray. Each physical subarray contributes $\frac{1}{8}$ of data to a logical subarray. For instance, an 8-byte Mat (black square in Figure 3) in logical subarray consists of 8-bit data chunks from 8 different physical subarrays as shown in Figure 3.

The size of a cache block is 64 bytes (i.e., 8 Mats) for RC-NVM. As shown in Figure 3, a cache block will be transferred from memory to row or column buffer if referenced in the corresponding address space.

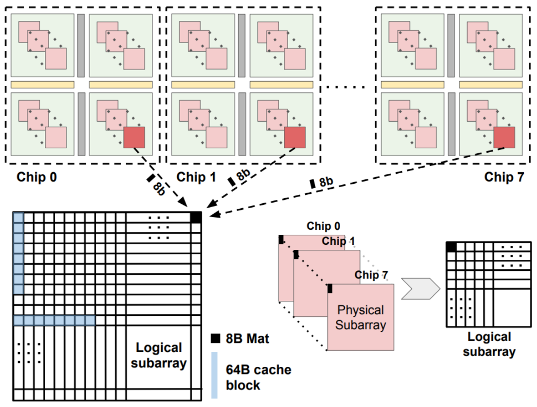


Figure 3: Physical and logical subarray

2.3 Observation: Cache Block Utilization

Given the access flexibility provided by the RC-NVM, graph systems can effectively avoid issuing unnecessary memory instructions. However, we observed that graph systems might suffer from the over-read issue where the utilization of some cache blocks is low. Due to the size mismatching between each source-destination pair (24B[2]) in a shard format and a cache block (64B), each cache block might contain two source-destination pairs. The over-read issue is caused when only one source-destination pair is used in an accessed cache block. That is, the utilization of the cache block is only about 50%. Moreover, due to the fact that graphs usually follow the power-law distribution, most vertices have very few in-edges and out-edges, and thus source-destination pairs belonging to irrelevant vertices have a high probability to be stored in the same cache block.

Generally, there are two different data placement strategies for placing source-destination pairs on RC-NVM, that is the row-first placement and the Z-ordering placement. We found that systems might suffer from the over-read issue no matter which placement strategy is adopted. The row-first placement places all pairs in a shard row-by-row, as shown in Figure 4(a). The row-first placement strategy offers high cache block utilization when a column-oriented access is issued, but suffers from the serious over-read issue while systems trigger a row-oriented access. On the other hand, the Z-ordering placement aims to place more pairs belonging to the same shard in the same cache block, as shown in Figure 4(b).

[2]In our system, we use 8 bytes to represent both source and destination indices, and 8 bytes to represent an edge value.

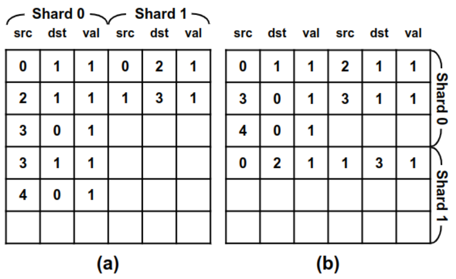


Figure 4: (a) Row-first placement strategy, and (b) Z-ordering placement strategy.

The Z-ordering placement strategy provides high cache block utilization when a row-oriented access is issued, but suffers from the serious over-read issue while systems trigger a column-oriented access. To illustrate our designs easily, we use the row-first placement in the reset of this paper. Please note that, with some modest adjustments, our solution can also support systems using the Z-ordering placement.

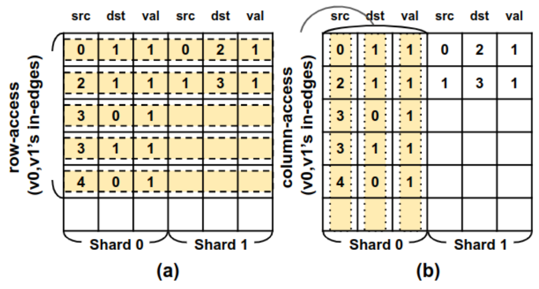

row-access (v0,v1's in-edges)
column-access (v0,v1's in-edges)
src dst val src dst val
Shard 0
Shard 1
(a)
(b)

Figure 5: Access in-edge weights of vertex 0, 1 in (a) row address space or in (b) column address space.

Based on our observations, selecting suitable memory instructions (i.e., row-oriented or column-oriented) with considering the graph data layout on RC-NVM could alleviate the over-read issue and thus improve the utilization of cache block. Assuming the graph system decides to read all in-edges associated with two vertices (i.e., v0, v1) from the graph in Figure 1(a). To simplify the explanations, we assume the cache block size is set to 48 bytes in this example. The system requires 5 row-oriented instructions (as shown in Figure 5(a)) or 3 column-oriented instructions (as shown in Figure 5(b)) to read out all data. Obviously, the cache block utilization is 50.0% and 83.3% while using row-oriented instructions and column-oriented instructions, respectively. In this case, systems can save 2 memory instructions if we decide to use column-oriented instructions to read out the required data.

2.4 Motivation

Aiming at improving the performance while accessing graph data stored in RC-NVM, we are mainly interested in how to improve the cache block utilization by carefully deciding on suitable memory instructions (i.e., row-oriented or column-oriented). *In contrast to previous designs which cannot effectively decide suitable memory instructions for different graph requests, this work is interested in proposing a hardware/software co-design solution to make the instruction decision by considering the graph data layout on RC-NVM.* The major technical challenges are on how to merge graph requests, which access the source-destination pairs located in the same cache block, and how to decide a cache-block-utilization-friendly memory instructions for serving each request issued from graph systems.

3 GRAPHRC

This section first demonstrates how GraphRC leverages the power of dual-addressing memory by proposing a vertex merging (VM) method that merges memory instructions of consecutive vertices for a better cache block utilization rate. Subsequently, we discuss how data dependency embedded in a graph affects the usage of VM and discover that the improvement obtained from VM is limited by the number of mergeable vertices in a graph. In order to further improve the performance, we propose an aggressive vertex merging (AVM) method that achieves more speedup by ignoring the dependency inherent in a graph.

3.1 Vertex Merging

In this section, we run a graph algorithm on GraphRC as an example to demonstrate how vertex merging improves cache block utilization among consecutive vertices. Furthermore, we identify the 4 restrictions on vertex merging and the consequences upon violation.

Generally speaking, a graph algorithm involves a sequence of in-edge and out-edge requests for each vertex. Without loss of generality, we assume a vertex first reads from its in-edges to gather information from its neighbors and then writes to its out-edges for results propagation. Other graph algorithms might have different patterns; however, similar results can be deduced by analogy.

The shard table in Figure 6(a) illustrates how the graph in Figure 1(a) is preprocessed into shard format and stored in a RC-NVM logical subarray. It contains *idx*, *src*, *dst*, *val* fields. The *idx* field shows the index of an edge within a shard, and the other three fields store necessary information for an edge. The *idx* field will not occupy any storage in memory; however, it is still listed for the ease of reading. The memory instruction table in Figure 6(b) lists the memory instructions generated by GraphRC accelerator. The *Type* field shows the memory instruction type in that clock cycle. Available options are R (row read), W (row write), CR (column read), CW (column write). The *Index* field shows the index range of a targeted memory region. The *Field* option shows which field will be accessed. Available options are: *src*, *dst*, *val*, and *all* (all three fields are accessed).

Graph data is preprocessed into shard format so in-edges of vertex 2, 3, 4 shall appear in shard 1 in Figure 6(a). In cycle $\underline{0}$ and $\underline{1}$, two column read (CR) instructions on *dst* and *val* fields retrieve the in-edges of vertex 2 from shard 1 as shown in Figure 6(b). The first column read (CR) instruction in cycle $\underline{0}$ finds edges with *dst* field equals to 2, and the second column read (CR) instruction in cycle $\underline{1}$ finds the corresponding in-edge weights for vertex 2. Subsequently, in-edges of vertex 3 and 4 are retrieved from shard 1 in cycle $\underline{3}$, $\underline{4}$, and $\underline{7}$ using a similar manner. Notice that in cycle $\underline{7}$, a column read instruction in *dst* field is required to figure out that there is no in-edge for vertex 4; therefore, a column read instruction in *val* field can be skipped. All three vertices (vertex 2, 3, 4) read the same memory region in shard 1. The in-edge requests of all

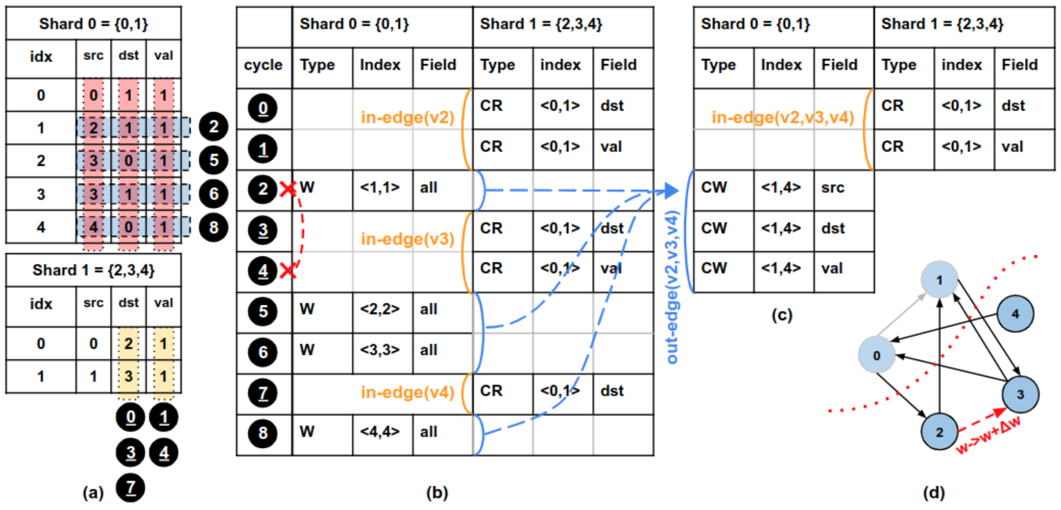


Figure 6: (a) Shard tables (b) Memory instruction table (only contains memory instructions for vertex 2,3,4) (c) Memory instruction table after vertex merging. (d) An intra-interval connection added between vertex 2 and 3.

three vertices can be shared and the corresponding column read instructions can be merged. As shown in Figure 6(b) and Figure 6(c), in-edge requests are merged for those vertices.

In cycle 2, one row write (W) instruction writes to out-edge of vertex 2 in shard 0. This instruction targets edges with *src* field equals to 2 in shard 0. Subsequently, out-edges of vertex 3 and 4 are written to shard 0 in cycle 5, 6, and 8. Out-edge requests for vertex 2, 3, and 4 form a 4-unit-tall and 3-unit-wide memory access region (blue) in Figure 6(a). This region requires 4 row-oriented cache blocks to cover and each row-oriented cache block only covers one source-destination pair. Alternatively, the same memory region can be covered with 3 column-oriented cache blocks in *src*, *dst*, and *val* fields (red). In that case, the cache block utilization rate will be improved as the number of cache blocks required to cover this region is reduced from 4 to 3. The geometric overview offered by a shard table demonstrates how RC-NVM improves cache block utilization by switching between address spaces. As for instructions on the memory instruction table, 4 row writes (W) instructions marked by dotted blue lines are reduced to 3 column writes (CW) instructions in Figure 6(c).

It is possible to reorder and regroup memory instructions of consecutive vertices such that in-edge requests are shared among consecutive vertices and out-edge requests form an enlarged access region in RC-NVM. A target memory region will be covered with cache blocks from another dimension to improve the overall cache block utilization rate. This technique that improves the cache block utilization rate by alternating between both address spaces for consecutive vertices is called vertex merging (VM). Performance comparison between two memory instruction tables in Figure 6(b) and Figure 6(c) shows that the overall cycle time is reduced from 9 to 5 (1.8x speedup) with the help of vertex merging.

Unfortunately, some vertices cannot be merged due to the following 4 reasons. Firstly, vertex merging only works when consecutive vertices belong to the same interval. Intervals are disjoint sets of vertices in a graph, and edges in a graph are partitioned into shards based on intervals. For instance, vertex 2, 3, 4 in Figure 6(d) belong to the same interval so edges with destination vertices fall into this interval are grouped into shard 1. In-edges of vertices from different intervals lie in different shards, and it is impossible to have a cache block that spans multiple shards. Secondly, only consecutive vertices can be merged. It is meaningless to merge separate vertices since their edge data exhibits poor locality. Thirdly, a vertex cannot be merged if previous k vertices have been merged. k is the maximum number of consecutive vertices that can be merged. Finally. vertex merging does not work on consecutive vertices that have intra-interval edge connections. Consider the example in Figure 6(d), vertex merging cannot be applied to vertex 2, 3, 4 if there exists an edge between vertex 2 and vertex 3.

This additional edge implies data dependency between vertex 2 writing to its out-edges, and vertex 3 reading from its in-edges. As shown in the dotted red line in Figure 6(b), this constraint enforces that the row write (W) instruction in cycle 2 must precede the column read (CR) instruction in cycle 4. It cannot be met if vertex merging is applied since vertex 3 reads from its in-edges before vertex 2 writes to its out-edges in Figure 6(c). Assume w is the original edge weight between vertex 2 and 3 as shown in Figure 6(d). Without vertex merging, vertex 2 should propagate its data to vertex 3 by writing $w + \Delta w$ to its out-edges. Vertex 3 will then compute its data based on $w+\Delta w$. However, if vertex merging is applied regardless of the dependency in between, vertex 3 will receive out-dated in-edge weight w while the correct one is $w+\Delta w$. The difference between w and $w + \Delta w$ might seems negligible for some algorithms but can significantly affect error-sensitive ones.

3.2 Aggressive Vertex Merging

For ranking-based graph algorithms people only care about the highest ranking (or perhaps top-N ranking) vertices in a graph, the exact outcome of each vertex is not an issue; however, the relative ranking order is more of a concern.

As discussed in the previous section, intra-interval connections limit the number of vertices that can be merged. For error-sensitive graph algorithms, the data dependency induced in the graph structure must be respected and that poses a theoretical limit for the speedup we can harvest from vertex merging. A vertex cannot be merged if it has intra-interval edges or if we reach the maximum number of vertices that can be merged. Now assume the maximum number is set to infinity, the number of vertices that can be merged will only depend on the intra-interval edge connections. That is to say, the theoretical speedup we can obtain from vertex merging depends on the structure of a graph.

Intra-interval edge connections within a graph limit our ability to merge memory instructions of vertices to gain speedup. Breaking such dependency results in unacceptable outcomes for error-sensitive algorithms but is considered tolerable for ranking algorithms. We believe that data dependencies of important vertices in a graph might have a larger impact on the algorithmic outcome compared to data dependencies of trivial vertices. To be more specific, we respect the constraints of important vertices and declare them as "unmergeable" while ignoring the constraints of trivial vertices. As mentioned in the example in Figure 6(d), an unmergeable vertex will receive outdated edge weights if it is eventually merged, and thus, generates inaccurate results. Thus, if a vertex is considered important, its data dependency will be respected and should not be merged to ensure minimal accuracy loss. On the contrary, data dependency of trivial vertices can be ignored so that it can speedup the performance through vertex merging. Trading accuracy for speedup is feasible for ranking algorithms such as PageRank [20] or ArticleRank [16].

Based on this observation, we propose aggressive vertex merging (AVM) that merges a vertex based on its importance instead of its inherent data dependency. There are lots of metrics to evaluate the importance of a vertex. Without loss of generality, we use Degree Centrality algorithm, which counts the total number of edges connected to a vertex, as an evaluation metric for the importance of a vertex. The adoption of Degree Centrality is reasonable because low-degree vertices usually access a small number of entries in a shard table and should be merged with others to improve cache block utilization. As for high degree vertices, which access many entries in a shard table, they don't need to "carpool" with other vertices to further improve cache block utilization.

3.3 Cost to Support VM and AVM

Apparently, the control logic for the graph accelerator needs to be modified to support both VM and AVM. The original control loop is redesigned to the one in Algorithm 1. A FIFO queue is introduced in the control loop to buffer vertices that can be merged. The queue is emptied if it encounters an unmergeable vertex or if the queue is full. Else, vertices will be buffered and wait to be dispatched. Queue size is an architectural parameter, and the relation between speedup and the size of a queue will be discussed in Section 4.

The dispatch subroutine in Algorithm 1 is explained in Algorithm 2. This function first issues read requests for all buffered vertices in queue, and all the data read from RC-NVM will be retained. Subsequently, after every vertex finishes its computation, results of all buffered vertices will be written back to RC-NVM.

Algorithm 1 GraphRC control loop for VM

```
Input: iteration ← maximum iteration count
Input: $G = (V, E)$ ← input graph
 1: procedure ControlLoop(iteration, G)
 2:     Queue queue;
 3:     for $itr$ ← 0 to iteration do
 4:         for each $v$ ← 0 to G.V do
 5:             if $v$.not_mergeable then
 6:                 Dispatch(queue)                ▷ $v$ cannot be merged
 7:             else
 8:                 queue.push($v$)
 9:             end if
10:             if queue.is_full then
11:                 Dispatch(queue)                ▷ queue is full
12:             end if
13:         end for
14:     end for
15: end procedure
```

Algorithm 2 Dispatch function

```
 1: procedure Dispatch(queue)
 2:     for each v ∈ queue do                     ▷ Read in-edges
 3:         v.read_in_edges()
 4:     end for
 5:     for each v ∈ queue do                     ▷ Compute
 6:         v.compute()
 7:     end for
 8:     for each v ∈ queue do                     ▷ Write out-edges
 9:         v.write_out_edges()
10:     end for
11:     queue.size ← 0                            ▷ Empty queue
12: end procedure
```

4 EXPERIMENTAL EVALUATION

4.1 Experimental Setup

In this section, we discuss the the experimental procedures, graph algorithms evaluated, and graph datasets used in the experiments.

4.1.1 Experimental Procedures. Experimental procedures consist of three parts: graph partition, memory trace generation, and performance evaluation. Firstly, we adopt the shard-based graph partition algorithm from GraphChi [13] by decoding its intermediate files. We then use the decoded files to simulate the graph processing flow in our in-house simulator written in C++. Secondly, our simulator takes shard data and vertex-centric graph algorithms as inputs and generates memory traces for the entire execution process. Finally, we evaluate the performance by sending cycle-accurate trace files to the RC-NVM simulator to obtain the overall execution time. All experiments are conducted on an Intel i7-8700 computer.

4.1.2 Graph Algorithms. All 6 graph algorithms implemented on our simulator follow the vertex-centric-like API defined by GraphChi [13]. PageRank [20] (PR), Connected Components (CC), Degree Centrality (DC) algorithms are modified from open-source online examples. ArticleRank [16] (AR), Breadth-First Search (BFS), and Single-Source Shortest Path (SSSP) algorithms are self-implemented and written in the most efficient way best to our knowledge [29].

4.1.3 Graph Datasets. We run all graph algorithms on four graph datasets provided by SNAP [15], as shown in Table 1. ego-Facebook (FB) and ego-Twitter (TW) are social network graphs. Wiki-Vote

Dataset name	# of Vertices	# of Edges
ego-Facebook (FB) [15]	4,039	88,234
Wiki-Vote (WV) [15]	7,115	103,689
ego-Twitter (TW) [15]	81,306	1,768,149
web-NotreDame (ND) [15]	325,729	1,497,134

Table 1: Graph datasets

(WV) is the voting network among Wikipedia contributors. web-NotreDame (ND) is the network topology of the University of Notre Dame website. The number of vertices and edges in each dataset is shown in Table 1.

	FB	WV	TW	ND
Mergeable vertices (%)	42.06%	80.46%	14.57%	68.97%

Table 2: Percentage of vertices that can be merged in different graph datasets.

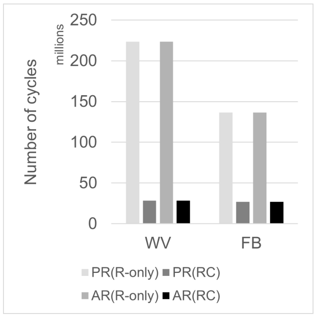
As illustrated in section 2, graph data will be preprocessed to a shard format and stored in RC-NVM. Vertices in a graph are first partitioned into intervals, and a vertex is mergeable if it does not have intra-interval edge connections. An intra-interval edge connection is defined as an edge connection between vertices located in the same interval. The percentage of vertices that can be merged for each graph dataset is shown in Table 2. A higher percentage in Table 2 indicates a graph has higher potential for vertex merging.

4.2 Performance

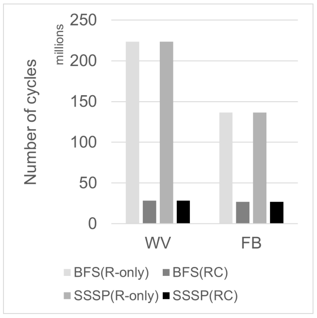
In this section, all graph algorithms are evaluated on graph datasets to compare the performance improvement obtained in different scenarios. This section is divided into 3 parts. The first part discusses the improvement offered by RC-NVM. The second part discusses the improvement offered by our VM method given the different number of vertices being merged. The third part evaluates the impact of AVM method and discusses how it surpasses VM method.

4.2.1 RC-NVM. In Figure 7, all graph algorithms are executed on all datasets for both traditional row-only memory and RC-NVM. The overall execution time of all 6 algorithms on WV and FB datasets are shown in Figure 7(a), Figure 7(b), and Figure 7(c), respectively. The overall execution time of all 6 algorithms on TW and ND datasets are shown in Figure 7(d), Figure 7(e), and Figure 7(f), respectively. As for traditional row-only memory, it is simulated by RC-NVM with dual-addressing ability disabled. Therefore, in-edge or out-edge requests will be converted to row-only addresses for traditional memory and converted to row and column-oriented addresses for RC-NVM. Note that all 6 graph algorithms are executed in a vertex-by-vertex manner and merging method has not been applied so far. Overall, RC-NVM helps reduce the execution time by at least 80.30% (at most 91.94% , on average 87.89%) among all datasets. The improvement is phenomenal and the reason behind can be best explained by the examples in Figure 5(a) and Figure 5(b).

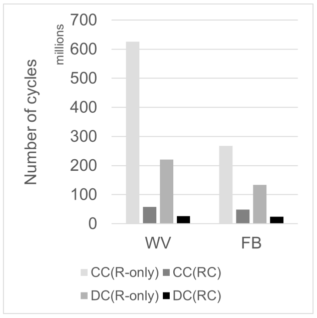
4.2.2 Vertex Merging. VM improves cache block utilization by reordering memory accesses of consecutive vertices. The merging mechanism can be implemented with a FIFO queue and the cost to support VM is explained in details in Section 3. The size of the queue controls the maximum number of vertices that can be



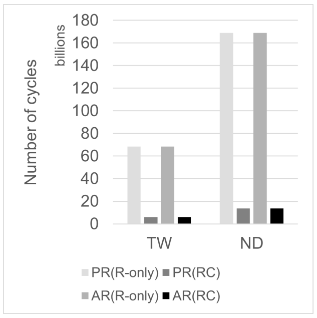
(a) PR and AR algo. on WV and FB datasets.



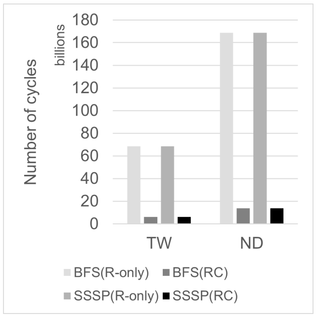
(b) BFS and SSSP algo. on WV and FB datasets.



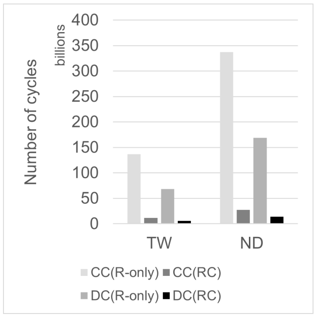
(c) CC and DC algo. on WV and FB datasets.



(d) PR and AR algo. on TW and ND datasets.



(e) BFS and SSSP algo. on TW and ND datasets.

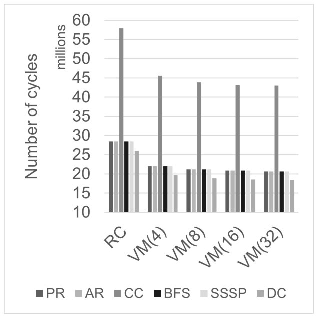


(f) CC and DC algo. on TW and ND datasets.

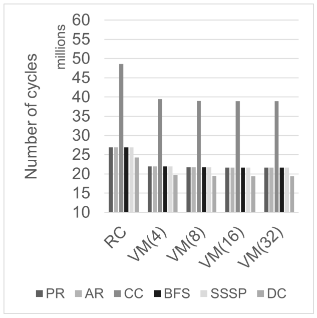
Figure 7: The performance comparisons between traditional row-only memory, and RC-NVM. R-only for traditional memory, and RC for RC-NVM.

merged in a graph. The effect of queue size on the overall performance is shown in Figure 8. The results in Figure 8(a), Figure 8(b), Figure 8(c), and Figure 8(d) show how VM improves the overall performance on WV, FB, TW, and ND datasets, respectively. When queue size is set to 4, at most 4 consecutive vertices can be merged, and the execution time of all algorithms is reduced by at least 14.61% (at most 52.78% , on average 24.24%) among all datasets. For all cases in Figure 8, the effectiveness of vertex merging plateaus to a limit as the queue size increases. The diminishing effect is caused by the data dependency embedded in the structure of a graph.

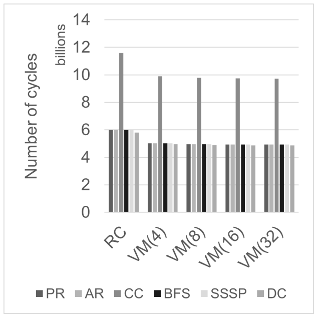
Recall that the effect of vertex merging is limited by the percentage of mergeable vertices in a graph. As the queue size increases, graphs with more mergeable vertices will gain more speedup. As



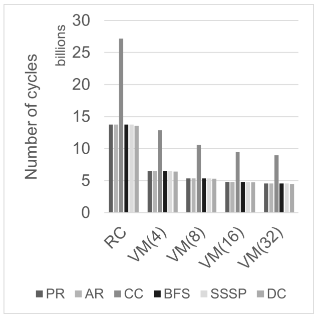
(a) VM applied to all algo. on WV datasets.



(b) VM applied to all algo. on FB datasets.

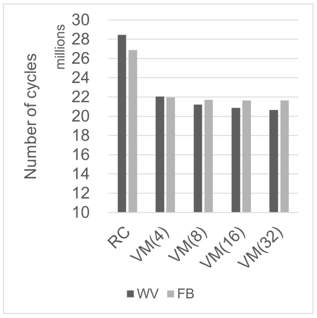


(c) VM applied to all algo. on TW datasets.

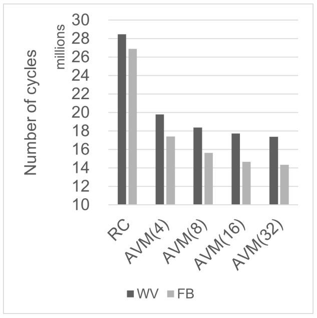


(d) VM applied to all algo. on ND datasets.

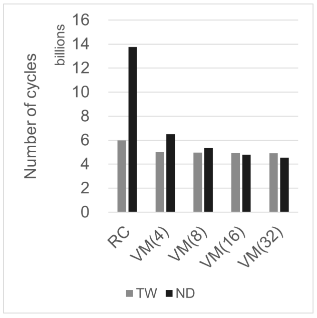
Figure 8: The effect of vertex merging (VM). Notice that RC stands for no vertex merging, and VM(*num*) means the queue size is set to *num*.



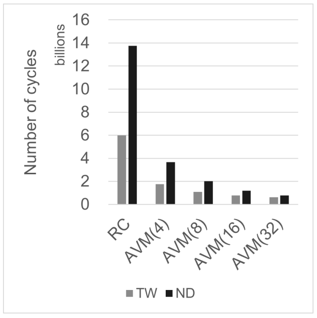
(a) VM applied to PR algo. on WV and FB datasets.



(b) AVM applied to PR algo. on WV and FB datasets.



(c) VM applied to PR algo. on TW and ND datasets.



(d) AVM applied to PR algo. on TW and ND datasets.

Figure 9: The effect of aggressive vertex merging (AVM) on PageRank algorithm. Notice that RC stands for no vertex merging, and AVM(*num*) means the queue size is set to *num*.

shown in Table 2, WV and ND datasets have 80.46% and 68.97% of mergeable vertices respectively. So, the impact of vertex merging is more significant for both datasets in Figure 8(a) and Figure 8(d).

4.2.3 Aggressive Vertex Merging. AVM ignores data dependency in graphs for better cache block utilization and is suitable for error-tolerant algorithms. In Figure 9(a) and Figure 9(b), we compare the performance of VM and AVM on WV and FB datasets. If the queue size is set to 4, AVM reduces execution time by 73.27% on WV dataset while VM only reduces 22.51%. It can be observed that AVM offers more speedup compared to VM as the size of the queue increases. In Figure 9(c) and Figure 9(d), the effectiveness of AVM is more significant as we compare the performance of AVM and VM on TW and ND datasets. Results show that AVM offers more speedup as queue size increases; however, it achieves more speedup at the expense of accuracy losses. For error-tolerant algorithms like PR and AR, the ordering of the top 20 vertices remains correct in all datasets when AVM is applied.

5 CONCLUSION

We present GraphRC, a graph accelerator that leverages the power of dual-addressing memory by mapping in-edge/out-edge requests to column/row-oriented memory accesses. The capability of dual-addressing memory greatly improves the performance of graph processing; however, some memory accesses still suffer from a low utilization rate due to the size mismatching between graph data and the cache block size. Therefore, we propose a vertex merging (VM) method that improves the cache block utilization rate by merging memory requests from consecutive vertices. VM reduces the execution time of all 6 graph algorithms on all 4 datasets by 24.24% on average. We then identify the data dependency inherent in a graph limits the usage of VM, and its effectiveness is bounded by the percentage of mergeable vertices. To overcome this limitation, we propose an aggressive vertex merging (AVM) method that outperforms VM by ignoring the data dependency inherent in a graph. AVM significantly reduces the execution time of ranking-based algorithms on all 4 datasets while preserving the correct ranking of the top 20 vertices.

6 ACKNOWLEDGMENTS

This work was supported in part by Academia Sinica under Grant AS-IA-111-M01, AS-GCS-110-08, and AS-CDA-107-M05. National Science and Technology Council under Grant 111-2223-E-001-001, 111-2221-E-001-013-MY3, 110-2221-E-006-084-MY3, and 109-2628-E-006-012-MY3. This work is also supported in part by Intelligent Manufacturing Research Center From the Featured Areas Research Center Program by the Ministry of Education, Taiwan (ROC).

REFERENCES

[1] Elia Ambrosi, Alessandro Bricalli, Mario Laudato, and Daniele Ielmini. 2019. Impact of oxide and electrode materials on the switching characteristics of oxide ReRAM devices. *Faraday Discuss.* 213 (2019), 87–98. Issue 0. https://doi.org/10.1039/C8FD00106E

[2] Renhai Chen, Zili Shao, and Tao Li. 2016. Bridging the I/O performance gap for big data workloads: A new NVDIMM-based approach. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 1–12. https://doi.org/10.1109/MICRO.2016.7783712

[3] Tseng-Yi Chen, Yuan-Hao Chang, Ming-Chang Yang, and Huang-Wei Chen. 2020. How to Cultivate a Green Decision Tree without Loss of Accuracy?. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design* (Boston, Massachusetts) *(ISLPED '20).* Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3370748.3406566

[4] Wei-Hao Chen, Kai-Xiang Li, Wei-Yu Lin, Kuo-Hsiang Hsu, Pin-Yi Li, Cheng-Han Yang, Cheng-Xin Xue, En-Yu Yang, Yen-Kai Chen, Yun-Sheng Chang, Tzu-Hsiang Hsu, Ya-Chin King, Chorng-Jung Lin, Ren-Shuo Liu, Chih-Cheng Hsieh, Kea-Tiong Tang, and Meng-Fan Chang. 2018. A 65nm 1Mb nonvolatile computing-in-memory ReRAM macro with sub-16ns multiply-and-accumulate for binary DNN AI edge processors. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC).* 494–496. https://doi.org/10.1109/ISSCC.2018.8310400

[5] Hsiang-Yun Cheng, Chun-Feng wu, Christian Hakert, Kuan-Hsun Chen, Yuan-Hao Chang, Jian-Jia Chen, Chia-Lin Yang, and Tei-Wei Kuo. 2021. Future Computing Platform Design: A Cross-Layer Design Approach. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE).* 312–317. https://doi.org/10.23919/DATE51398.2021.9474229

[6] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, Jaehong Kim, Jaechun Park, Ki-Whan Song, Ki-Tae Park, Sangyeun Cho, Hwaseok Oh, Daniel D.G. Lee, Jin-Hyeok Choi, and Jaeheon Jeong. 2018. A flash memory controller for 15μs ultra-low-latency SSD using high-speed 3D NAND flash with 3μs read time. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC).* 338–340. https://doi.org/10.1109/ISSCC.2018.8310322

[7] 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).* 27–39. https://doi.org/10.1109/ISCA.2016.13

[8] Chien-Chung Ho, Wei-Chen Wang, Szu-Yu Chen, Yung-Chun Li, and Kun-Chi Chiang. 2022. RAM: Exploiting Restrained and Approximate Management for Enabling Neural Network Training on NVM-Based Systems. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing* (Virtual Event) *(SAC '22).* Association for Computing Machinery, New York, NY, USA, 116–123. https://doi.org/10.1145/3477314.3507090

[9] Yu Ting Ho, Chun-Feng Wu, Ming-Chang Yang, Tseng-Yi Chen, and Yuan-Hao Chang. 2019. Replanting Your Forest: NVM-friendly Bagging Strategy for Random Forest. In *2019 IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA).* 1–6. https://doi.org/10.1109/NVMSA.2019.8863525

[10] Yao-Wen Kang, Chun-Feng Wu, Yuan-Hao Chang, Tei-Wei Kuo, and Shu-Yin Ho. 2020. On Minimizing Analog Variation Errors to Resolve the Scalability Issue of ReRAM-Based Crossbar Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3856–3867. https://doi.org/10.1109/TCAD.2020.3012250

[11] Akifumi Kawahara, Ryotaro Azuma, Yuuichirou Ikeda, Ken Kawai, Yoshikazu Katoh, Kouhei Tanabe, Toshihiro Nakamura, Yoshihiko Sumimoto, Naoki Yamada, Nobuyuki Nakai, Shoji Sakamoto, Yukio Hayakawa, Kiyotaka Tsuji, Shinichi Yoneda, Atsushi Himeno, Ken-ichi Origasa, Kazuhiko Shimakawa, Takeshi Takagi, Takumi Mikawa, and Kunitoshi Aono. 2012. An 8Mb multi-layered cross-point ReRAM macro with 443MB/s write throughput. In *2012 IEEE International Solid-State Circuits Conference.* 432–434. https://doi.org/10.1109/ISSCC.2012.6177078

[12] Gokul Krishnan, Jubin Hazra, Maximilian Liehr, Xiaocong Du, Karsten Beckmann, Rajiv V. Joshi, Nathaniel C. Cady, and Yu Cao. 2021. Design Limits of In-Memory Computing: Beyond the Crossbar. In *2021 5th IEEE Electron Devices Technology & Manufacturing Conference (EDTM).* 1–3. https://doi.org/10.1109/EDTM50988.2021.9421057

[13] 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).* USENIX Association, Hollywood, CA, 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[14] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T.W. Keller. 2003. Energy management for commercial servers. *Computer* 36, 12 (2003), 39–48. https://doi.org/10.1109/MC.2003.1250880

[15] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[16] Jiang Li and Peter Willett. 2009. ArticleRank: a PageRank-based alternative to numbers of citations for analysing citation networks.

[17] Shuo Li, Nong Xiao, Peng Wang, Guangyu Sun, Xiaoyang Wang, Yiran Chen, Hai Helen Li, Jason Cong, and Tao Zhang. 2019. RC-NVM: Dual-Addressing Non-Volatile Memory Architecture Supporting Both Row and Column Memory Accesses. *IEEE Trans. Comput.* 68, 2 (2019), 239–254. https://doi.org/10.1109/TC.2018.2868368

[18] Taozhong Li, Naifeng Jing, Zhigang Mao, and Yiran Chen. 2022. A Hybrid-Grained Remapping Defense Scheme Against Hard Failures for Row-Column-NVM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 6 (2022), 1842–1854. https://doi.org/10.1109/TCAD.2021.3097288

[19] Ting-Shan Lo, Chun-Feng Wu, Yuan-Hao Chang, Tei-Wei Kuo, and Wei-Chen Wang. 2021. Space-efficient Graph Data Placement to Save Energy of ReRAM Crossbar. In *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED).* 1–6. https://doi.org/10.1109/ISLPED52811.2021.9502482

[20] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[21] Mohit Saxena and Michael M. Swift. 2010. FlashVM: Virtual Memory Management on Flash. In *2010 USENIX Annual Technical Conference (USENIX ATC 10).* USENIX Association. https://www.usenix.org/conference/usenix-atc-10/flashvm-virtual-memory-management-flash

[22] 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. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA).* 14–26. https://doi.org/10.1109/ISCA.2016.12

[23] Daniel Waddington and Jim Harris. 2018. Software Challenges for the Changing Storage Landscape. *Commun. ACM* 61, 11 (oct 2018), 136–145. https://doi.org/10.1145/3186331

[24] Peng Wang, Shuo Li, Guangyu Sun, Xiaoyang Wang, Yiran Chen, Hai Li, Jason Cong, Nong Xiao, and Tao Zhang. 2018. RC-NVM: Enabling Symmetric Row and Column Memory Accesses for In-memory Databases. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 518–530. https://doi.org/10.1109/HPCA.2018.00051

[25] Wei-Chen Wang, Yuan-Hao Chang, Tei-Wei Kuo, Chien-Chung Ho, Yu-Ming Chang, and Hung-Sheng Chang. 2019. Achieving Lossless Accuracy with Lossy Programming for Efficient Neural-Network Training on NVM-Based Systems. *ACM Transactions on Embedded Computing Systems* 18, 5s, Article 68 (oct 2019), 22 pages. https://doi.org/10.1145/3358191

[26] Chun-Feng Wu, Yuan-Hao Chang, Ming-Chang Yang, and Tei-Wei Kuo. 2020. Joint Management of CPU and NVDIMM for Breaking Down the Great Memory Wall. *IEEE Trans. Comput.* 69, 5 (2020), 722–733. https://doi.org/10.1109/TC.2020.2964254

[27] Chun-Feng Wu, Yuan-Hao Chang, Ming-Chang Yang, and Tei-Wei Kuo. 2020. When Storage Response Time Catches Up With Overall Context Switch Overhead, What Is Next? *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 4266–4277. https://doi.org/10.1109/TCAD.2020.3012322

[28] Chun-Feng Wu, Ming-Chang Yang, Yuan-Hao Chang, and Tei-Wei Kuo. 2018. Hot-Spot Suppression for Resource-Constrained Image Recognition Devices With Nonvolatile Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2567–2577. https://doi.org/10.1109/TCAD.2018.2858459

[29] Da Yan, Bu Yingyi, Yuanyuan Tian, and Amol Deshpande. 2017. *Big Graph Analytics Platforms.* 27–28 pages.

[30] Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. 2022. Practicably Boosting the Processing Performance of BFS-like Algorithms on Semi-External Graph System via I/O-Efficient Graph Ordering. In *20th USENIX Conference on File and Storage Technologies (FAST 22).* USENIX Association, Santa Clara, CA, 381–396. https://www.usenix.org/conference/fast22/presentation/yang

[31] Xiaoxuan Yang, Brady Taylor, Ailong Wu, Yiran Chen, and Leon O. Chua. 2022. Research Progress on Memristor: From Synapses to Computing Systems. *IEEE Transactions on Circuits and Systems I: Regular Papers* 69, 5 (2022), 1845–1857. https://doi.org/10.1109/TCSI.2022.3159153

[32] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15).* USENIX Association, Santa Clara, CA, 45–58. https://www.usenix.org/conference/fast15/technical-sessions/presentation/zheng