GraphS: A Graph Processing Accelerator Leveraging SOT-MRAM

Shaahin Angizi*, Jiao Sun†, Wei Zhang† and Deliang Fan*
*Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816
†Department of Computer Science, University of Central Florida
Email: angizi@knights.ucf.edu, dfan@ucf.edu

***Abstract*—In this work, we present GraphS architecture, which transforms current Spin Orbit Torque Magnetic Random Access Memory (SOT-MRAM) to massively parallel computational units capable of accelerating graph processing applications. GraphS can be leveraged to greatly reduce energy consumption dealing with underlying adjacency matrix computations, eliminating unnecessary off-chip accesses and providing ultra-high internal bandwidth. The device-to-architecture co-simulation for three social network data-sets indicate roughly 3.6× higher energy-efficiency and 5.3× speed-up over recent ReRAM crossbar. It achieves ∼4× higher energy-efficiency and 5.1× speed-up over recent processing-in-DRAM acceleration methods.**

I. Introduction

Achieving high bandwidth of graph processing in Von-Neumann platforms suffers from different challenges [1], such as long memory access latency, significant congestion at I/Os, huge data communication energy and large leakage power consumption for storing graph parameters in the volatile memory that lead to over 90% bandwidth degradation on CPU-DRAM hierarchy [2]. In the last two decades, Processing-in-Memory (PIM), as a potentially viable way to solve the memory wall challenge, have been put forward [3], [4]. The key idea of PIM is to embed or realize logic units within memory to process data by leveraging the inherent parallel computing mechanism and exploiting large internal memory bandwidth. It could lead to remarkable savings in off-chip data communication energy and latency. PIM architectures ideally should be capable of performing bulk bit-wise operations which is needed in many graph processing applications [5]. However, this has been limited to basic logic operations such as `AND`, `OR` and `XOR` so far [5], [6], which are not necessarily applicable to a wide variety of tasks except by imposing multi-cycle operations to realize specific functions such as addition [4], [7].

The proposals for exploiting SRAM-based [8], [9] PIM architectures can be found in recent literature. However, PIM in context of main memory (DRAM- [3], [4], [10]) has drawn much more attention mainly due to larger memory capacities and off-chip data transfer reduction as opposed to SRAM-based PIM. However, existing DRAM-based PIM architectures have major shortcomings, e.g., high refresh/leakage power, multi-cycle logic operations, operand data overwritten, operand locality, etc. Ambit [4] shows DRAM-based graph processing acceleration by realizing a majority function between every three rows and so can implement 2-input logic after saving operand data in reserved rows to avoid data-overwritten. GraphH [1] and Graphpim [11] present new designs based on Hybrid Memory Cube (HMC) to accelerate large-scale graph processing tasks at architectural level.

The PIM architectures have recently become even more popular when integrating with emerging Non-Volatile Memory (NVM) technologies, such as Resistive RAM (ReRAM) [6]. ReRAM offers more packing density ($\sim 2-4\times$) than DRAM, and hence appears to be competitive alternatives to DRAM. However, it still suffers from slower and more power hungry writing operations than DRAM [12]. Spin-Orbit Torque Magnetic Random Access Memory (SOT-MRAM) [13] technology is other promising high performance candidate for both last level cache and main memory, due to its low switching energy, non-volatility, superior endurance, excellent retention time, high integration density and compatibility with CMOS technology. Meanwhile, MRAM technology is undergoing the process of commercialization [14]. Thus, in-memory graph processing accelerators in the context of different NVMs, without sacrificing memory capacity, are of growing interest on graph processing tasks. For instance, Pinatubo [5] presents a general PIM architecture for NVM-based graph processing, where each sub-array is capable of performing bulk `AND/OR/XOR` functions thanks to modified decoder and memory sense amplifiers. MPIM [6] consists of multiple ReRAM crossbars and provides the same bit-wise operations with modified analog sense amplifiers.

From graph processing algorithm perspective, network topology analysis can help us better understand the intricate connectivity of complex networks in practical problems. For instance, degree centrality is often used to measure the importance of a vertex. In social networks, people with more connections tend to have more significant influence in the community. The matching index is another basic topology parameter characterizes the similarity between two vertices in a network. It measures the ratio of common neighbors for pair of vertices. Evaluation of these network properties plays an essential part in potential applications, such as social network analysis and traffic flow control. The main ***goal*** of this paper is to develop a non-volatile, parallel and energy-efficient PIM architecture that could simultaneously work as memory and realize a high performance accelerator for such data-intensive graph processing applications. The main contributions of this paper are summarized as follows: (1) We propose a novel SOT-MRAM in-memory accelerator, *GraphS*, based on set of novel microarchitectural and circuit-level schemes that position *GraphS* as a massive data-parallel computational unit with negligible area overhead. (2) We provide case studies of how important graph processing workloads can be partitioned mapped to our architecture and how they can benefit from it. (3) We evaluate our proposed scheme using a variety of real-world social network graph data compared with other state-of-the-art acceleration solutions i.e. DRAM, HMC, ReRAM, STT-MRAM, and GPU.

II. GraphS Architecture

A. Computational sub-arrays

GraphS is designed to be an independent high-performance and energy-efficient accelerator based on main memory architecture. The main memory chip is basically divided into

978-3-9819263-2-3/DATE19/©2019 EDAA

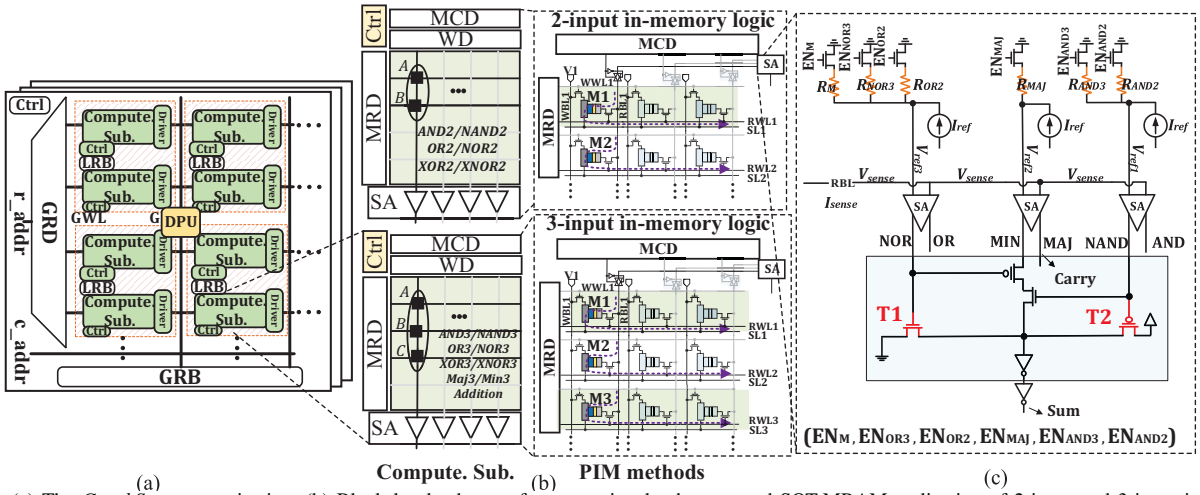

Fig. 1. (a) The *GraphS* mat organization, (b) Block level scheme of computational sub-array and SOT-MRAM realization of 2-input and 3-input in-memory logic methods, (c) Reconfigurable SA.

multiple Banks. Banks within the same chip typically share I/O, buffer and banks in different chips working in a lock-step manner. Each bank consists of multiple memory matrices (mats). The general mat organization of *GraphS* is shown in Fig. 1a. Each mat consists of multiple computational memory sub-arrays connected to a Global Row Decoder (GRD) and a shared Global Row Buffer (GRB). According to the application type and physical address of operands within memory, *GraphS* Controller (Ctrl) is able to configure the sub-arrays to perform data-parallel inter- and intra-sub-array computations. Every two sub-arrays share a Local Row Buffer (LRB) and there is a Digital Processing Unit (DPU) in each mat to further process the data (if necessary) in specific applications as will be discussed later.

Fig. 1b depicts the presented PIM sub-array architecture based on SOT-MRAM. This architecture mainly consists of Write Driver (WD), Memory Row Decoder (MRD), Memory Column Decoder (MCD), reconfigurable Sense Amplifier (SA), and can be adjusted by Ctrl unit to work in dual mode that perform both memory write/read and bit-line computing (using two distinct methods). SOT-MRAM device is a composite structure of spin Hall metal (SHM) and Magnetic Tunnel Junction (MTJ) [13]. The resistance of MTJ with parallel magnetization in both magnetic layers (data-'0') is lower than that of MTJ with anti-parallel magnetization (data-'1'). Each SOT-MRAM cell located in computational sub-arrays is associated with the Write Word Line (WWL), Read Word Line (RWL), Write Bit Line (WBL), Read Bit Line (RBL), and Source Line (SL) to perform operations based on reconfigurability of memory SAs.

The key idea to perform memory read and bit-line computing in *GraphS* is to choose different thresholds (references) when sensing the selected memory cell(s). The proposed reconfigurable SA, as depicted in Fig. 1c, consists of three sub-SAs and totally six reference-resistance branches that can be selected by enable bits (EN_M, EN_{OR3}, EN_{OR2}, EN_{MAJ}, EN_{AND3}, EN_{AND2}) by the sub-array's Ctrl to realize the memory and computation schemes as tabulated in Table I. Such reconfigurable SA could implement memory read and one-threshold based logic functions only by activating one enable at a time e.g. by setting EN_{AND2} to '1', 2-input AND/NAND logic can be readily implemented between operands located in the same bit-line. Meanwhile, by activating two or three enables at a time, two or three logic functions can be simultaneously implemented and further used to generate complex logic functions like XOR3/XNOR3, as explained accordingly.

TABLE I. CONFIG. OF ENABLE BITS FOR DIFFERENT FUNCTIONS.

Ops.	read	OR2/ NOR2	AND2/ NAND2	MAJ/ MIN	OR3/ NOR3	AND3/ NAND3	Add/ XOR3/XNOR3 XOR2/XNOR2
EN_M	1	0	0	0	0	0	0
EN_{OR2}	0	1	0	0	0	0	0
EN_{AND2}	0	0	1	0	0	0	0
EN_{OR3}	0	0	0	0	1	0	1
EN_{AND3}	0	0	0	0	0	1	1
EN_{MAJ}	0	0	0	1	0	0	1

Memory Mode: To *write* a bit in any of the SOT-MRAM cells, e.g. in the cell of 1st row and 1st column, write current should be injected through the heavy metal substrate of SOT-MRAM. To activate this write current path, WWL1 is activated by MRD and SL1 is grounded, while all the other lines are kept floating. Now, in order to write '1' (/'0'), the WD (V1) connected to WBL1 is set to positive (/negative) write voltage. This allows sufficient charge current flows from V1 to ground (/ground to V1), leading to MTJ resistance in High-R_{AP} (/Low-R_P). For typical memory *read*, a read current flows from the selected SOT-MRAM cell to ground, generating a sense voltage (V_{sense}) at the input of SA, which is compared with memory mode reference voltage activated by EN_M ($V_{sense,P}<V_{ref,M}<V_{sense,AP}$). Now, if the path resistance is higher (/lower) than R_M (memory reference resistance), i.e. R_{AP} (/R_P), then the SA produces High (/Low) voltage indicating logic '1' (/'0'). The idea of voltage comparison for memory read is shown in Fig. 2a. Based on the memory mode, we develop Fast Row Copy (FRC) mechanism that needs a consecutive memory read and write operations. In the first half-cycle, the source row is activated by sub-array's MRD and readout to LRB; in the second half-cycle, the data stored in buffer is written back to the destination row. It is noteworthy that FRC can be readily used in mat and bank levels considering inter-component's buffer (GRB) to accelerate copy operation in *GraphS*'s sub-components.

Bit-line Computing Mode: The computational sub-array of *GraphS* is designed to perform bulk bit-wise in-memory logic operations between two or three operands located in the same bit-line. In the 2-input in-memory logic method (*IML2x*), every two bits stored in the identical column can be selected and sensed simultaneously employing the MRD [5], as depicted in Fig. 1b. Then, the equivalent resistance of such

parallel connected cells and their cascaded access transistors are compared with a programmable reference by SA. Through selecting different reference resistances (R_{AND2}, R_{OR2}), the SA can perform basic 2-input in-memory Boolean functions (i.e. AND and OR), e.g. to realize AND operation, R_{ref} is set at the midpoint of $R_{AP}//R_P$ ('1','0') and $R_{AP}//R_{AP}$ ('1','1'). Consider the data organization shown in Fig. 1b L.H.S., where A and B operands correspond to M1 and M2 memory cells, respectively, 2-input in-memory logic method generates AB after SA in a single memory cycle. The idea of voltage comparison between V_{sense} and V_{ref} for IML2x is shown on Fig. 2b. It is worth pointing out that only one sub-SA is used during one-threshold logic operations to reduce the power consumption of sensing. Owing to the complementary outputs of sub-SAs, the reconfigurable SA can also provide 2-input NOR, NAND functions.

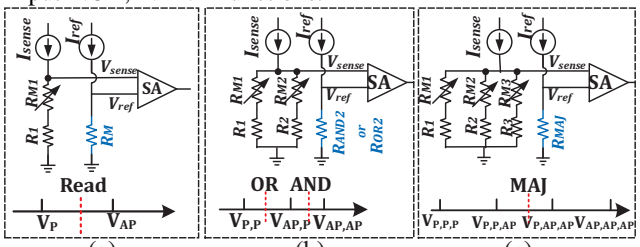

Fig. 2. The idea of voltage comparison between V_{sense} and V_{ref} for (a) memory read, (b) IML2x, (c) IML3x.

In the 3-input in-memory logic method (*IML3x*), every three cells located in an identical column can be selected by MRD and sensed simultaneously to realize 3-input majority/minority functions (`Maj/Min`) in a single sensing cycle. Consider the data organization shown in Fig. 1b where A, B and C operands correspond to M1, M2 and M3 memory cells, respectively, the computational sub-array can perform $AB + AC + BC$ Boolean function by setting EN_{MAJ} to '1'. As shown in Fig. 2c, to perform `MAJ` operation, R_{MAJ} is set at the midpoint of $R_P//R_P//R_{AP}$ ('0','0','1') and $R_P//R_{AP}//R_{AP}$ ('0','1','1'). In order to validate the variation tolerance of the sensing circuit, we have performed Monte-Carlo simulation with 10000 trials. A $\sigma = 2\%$ variation is added to the Resistance-Area product (RA_P), and a $\sigma = 5\%$ process variation is added on the Tunneling MagnetoResistive (TMR). The simulation result of sense voltage (V_{sense}) distributions in Fig. 3 shows the sense margin for conventional memory read, two fan-in in-memory logic and 3 fan-ins sense-based operation. It can be seen that sense margin gradually reduces when increasing the number of fan-ins (selected SOT-MRAM cells for computation). To avoid logic failure and guarantee the SA output's reliability, we have limited the number of sensed cells to three. Note that, such sense margin could be even improved by increasing the sense current, but by sacrificing the operation's energy-efficiency. Parallel computing/read is implemented by using one SA per bit-line.

In addition to the above-mentioned operations, *GraphS*'s sub-array can perform addition/subtraction (`add/sub`) operation quite efficiently. With a careful observation on the Full-Adder (FA) truth table, we observe that in six out of eight possible input combinations, Sum output can be directly obtained by inverted Carry signal. Keep this fact in mind that FA's Carry can be resulted from `MAJ` function, the proposed reconfigurable SA can implement such Sum output readily

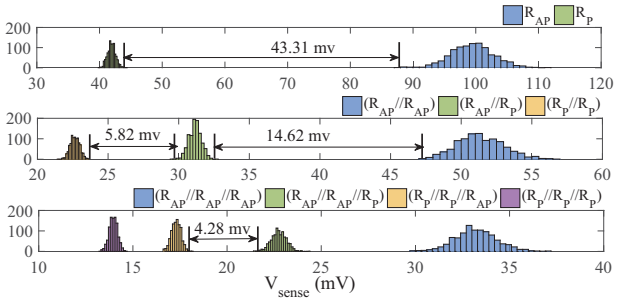

Fig. 3. Monte-Carlo simulation of V_{sense} distribution for (top) memory read operation, and bit-line computing (middle) IML2x (down) IML3x.

by `MIN` (majority-not) function inspired by [15]. As shown in Fig. 1c, the Sum signal is directly connected to the `MIN` output. However, for two extreme cases, i.e. (0,0,0) and (1,1,1), the `MIN` signal is disconnected and Sum can be respectively implemented by `NOR3` (T1:ON, T2:OFF $\rightarrow$ Sum=0) and `NAND3` functions (T1:OFF, T2:ON $\rightarrow$ Sum=1). This is realized by adding two pass transistors in the `MIN` function path. Note that, considering the fact that Sum output is the `XOR3` function, the proposed reconfigurable SA can also implement 2-input and 3-input `XOR/XNOR` functions, without imposing additional XOR gates like previous works [4], [9]. Now, assume A, B and C as input operands (in Fig. 1b), IML3x can generate Sum(/Difference) and Carry(/Borrow) bits in a single cycle. To the best of our knowledge, the design proposed here is the first PIM which can directly implement in-memory addition in a single memory cycle, where for example Ambit [4] and MPIM [6] impose over 4 cycles.

B. System integration

While *GraphS* is meant to be an independent high-performance and energy-efficient accelerator, we need to expose it to programmers and system-level libraries to utilize it. From a programmer perspective, *GraphS* is more of a third party accelerator that can be connected directly to the memory bus or through PCI-Express lanes rather than a memory unit, thus it is integrated similar to that of GPUs. Therefore, a virtual machine and ISA for general-purpose parallel thread execution need to be defined similar to PTX [16] for NVIDIA. Accordingly, the programs are translated at install time to the *GraphS* hardware instruction set tabulated in Table II. The micro and control transfer instructions are not shown here.

TABLE II. THE BASIC INSTRUCTIONS OF *GraphS*.

opcode		operation	function
FRC		$B \leftarrow A$	Copy row A to Row B
IML2x	IML21	$A.B$	`AND2/NAND2`
	IML22	$A+B$	`OR2/NOR2`
IML3x	IML31	$A.B.C$	`AND3/NAND3`
	IML32	$A+B+C$	`OR3/NOR3`
	IML33	$AB + AC + BC$	`MAJ/MIN`
	IML34	$A \oplus B$	`XOR2/XNOR2`
	IML35	$A \oplus B \oplus C$	`XOR3/XNOR3`
	IML36	Sum/Carry	`add/sub`

The *GraphS* commands/instructions can be directly copied/written to a predefined memory-mapped address ranges, e.g., defined in the memory type range registers (MTRRs), or programmed through writing to Memory-Mapped I/O regions that are allocated through a simple device driver to do initialization/cleanup for required software memory structures. Note that the first approach can potentially bring more performance gains compared to the later one; accessing *GraphS* as an I/O device can incur significant overheads due to interrupts and page faults (in case of shared memory model). In contrast,

memory-mapped *GraphS* scheme can cause major contentions in the memory bus in case the processor is executing memory-intensive applications simultaneously. We leave choosing the scheme of integrating *GraphS* to system architects based on their workloads and usecases. In both schemes for integrating *GraphS*, the commands/instructions that *GraphS* architecture accept is similar and based on the ISA.

III. Hardware Mapping

The *GraphS*'s parallel operations can be readily used to accelerate a wide variety of graph processing tasks. For the sake of limited space, we briefly explain two widely-used tasks so-called *degree centrality* and *matching index*.

A. Degree centrality

One of the most important graph processing tasks is degree centrality. This task deals with massive number of `add` operation which basically counts the number of valid links connected to a vertex. Fig. 4 shows an intuitive example of hardware mapping and acceleration of such operation performed by *GraphS* for a small graph. Initially, the designated graph is converted to adjacency matrix and mapped to consecutive rows of *GraphS*'s sub-arrays. Now, in the first half-cycle, every three rows are activated through RWLs sequentially (here, step (1) and (2)) to perform parallel `add` operation (IML36) and generate initial Carry (C) and Sum (S) bits. In the second half-cycle, the results are written back to the memory reserved space. Then, next steps ((3) and (4)) only deal with multi-bit addition of resultant data starting bit-by-bit from the LSBs of the two words continuing towards MSBs. There are 2 cycles for every bit-position computation. In the first half-cycle of (3), 2 RWLs (accessing to LSBs of 6 elements) and one RWL (accessing the reserved row initialized by zero) are enabled to generate the sum and carry. The SAs use these 3 words to generate sum and carry employing IML36. During second half-cycle, two WWLs are activated to save back the sum and carry bits. This process continues to MSB and concluded after $2 \times m$ cycles, where m is number of bits in elements. At the end the degree of each vertex is stored in memory (e.g. 4 determines the degree of vertex 1).

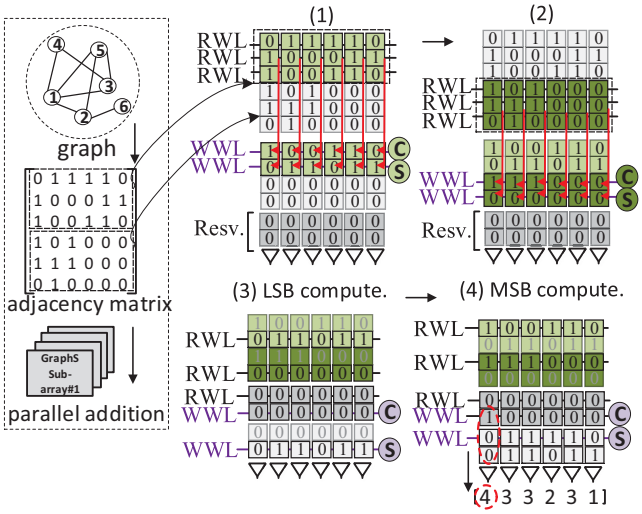

Fig. 4. *GraphS*'s mapping and acceleration for `add`-based graph processing operations. Here we take degree centrality computation as an example.

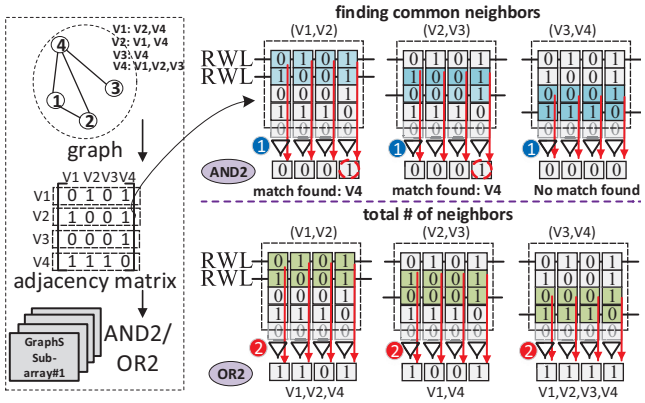

Fig. 5. *GraphS*'s mapping and acceleration for finding matching index.

B. Matching index

The matching index $M_{i,j}$ quantifies the "similarity" between two vertices based on the number of common neighbors shared by vertices as ($\frac{\sum \text{common neighbors}}{\sum \text{total number of neighbors}}$). The main task here is to find the common and total number of neighbors which can be carried out and accelerated by *GraphS*. Fig. 5 provides a straightforward example to elucidate the mapping and acceleration method of *GraphS*. Initially, the sample four-vertex network is converted to adjacency matrix and stored in 4 consecutive rows of sub-array. To find the common neighbors of two particular vertices (e.g. V1, V2), *GraphS* performs parallel `AND2` (IML21) on the rows and SA's outputs determine the matches (here, V4). In addition, the total number of neighbors is found by performing `OR2` (IML22) operation on the same rows. Then, *GraphS*'s `add` operation can readily process the summation operation as explained earlier. Afterwards, DPU is utilized to perform the division operation to generate corresponding index matrix.

C. Data partitioning and allocation

Real world graph consists of millions of vertices and edges that need to be processed. To efficiently map such graphs into *GraphS* architecture, graph partitioning methods are needed. Here, we adopt interval-block partitioning method to balance workloads of each *GraphS*'s chip and maximize parallelism. We use hash-based method [1] to split the vertices into M intervals and then divide edges into M^2 blocks as shown in Fig. 6. Now each block is allocated a specific chip and accordingly mapped to its sub-arrays. Considering an N-vertex sub-graph that needs to be mapped to a *GraphS* chip with N_s activated sub-arrays with the size of $x \times y$, each sub-array can process n vertices ($n \leq f | n \in N, f = min(x, y)$). Therefore, the number of sub-arrays for processing an N-vertex sub-graph can be formulated as, $N_s = \left\lceil \frac{N}{f} \right\rceil$.

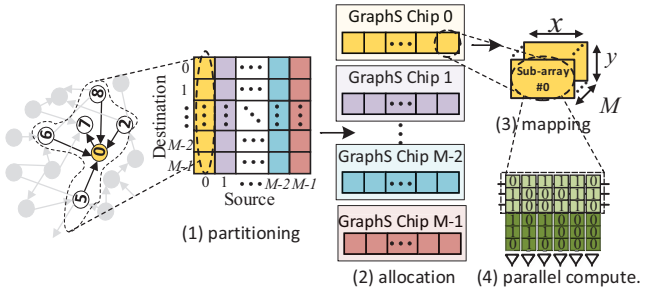

Fig. 6. Graph partitioning and allocation of *GraphS* accelerator.

IV. Performance Evaluations

In this section, we compare *GraphS* with other possible graph processing acceleration solutions based on DRAM, HMC, ReRAM, STT-MRAM and GPU. We configure the *GraphS*'s memory sub-array with 512 rows and 256 columns, 4×4 mats (with 1/1 as row/column activation) per bank organized in H-tree routing manner, 16×16 banks (with 1/1 as row/column activation) and 512Mb total capacity. While the computation is mainly performed on 256×256-bit sub-arrays, 256 rows are considered as reserved rows. It is obvious that enlarging the chip area brings a higher performance for *GraphS* and other PIM designs due to the increased number of computational sub-arrays, though the die size directly impacts the chip cost. Therefore, an identical physical memory size (512Mb) is considered for all implementations henceforth.

To evaluate the performance of accelerators, we take three social network data-sets as tabulated in Table III. Then, we map and run 3 graph processing tasks i.e. degree centrality, matching index and Breadth First Search (BFS) on them that seek most of *GraphS*'s operations.

TABLE III. Social Network data-sets.

Dataset	Nodes	Edges	Graph Information
ego-Facebook	4,039	88,234	profiles & friends lists from Facebook [17]
dblp-2010	326,186	1,615,400	scientific collaboration network
amazon-2008	735,323	5,158,388	similarity among books reported by Amazon store

A. Accelerators' setup

GraphS: To assess the performance of *GraphS* as a new PIM platform, a comprehensive device-to-architecture evaluation framework along with two in-house simulators are developed. First, at the device level, we jointly use the Non-Equilibrium Green's Function (NEGF) and Landau-Lifshitz-Gilbert (LLG) with spin Hall effect equations to model SOT-MRAM bit-cell [13]. For the circuit level simulation, a Verilog-A model of 2T1R SOT-MRAM device is developed to co-simulate with the interface CMOS circuits in Cadence Spectre and SPICE. 45nm North Carolina State University (NCSU) Product Development Kit (PDK) library [18] is used in SPICE to verify the proposed design and acquire the performance. Second, an architectural-level simulator is built based on NVSim [19]. Based on the device/circuit level results, our simulator can alter the configuration files (.cfg) corresponding to different array organization and report performance metrics for PIM operations. The controllers and add-on circuits are synthesized by Design Compiler [20] with an industry library. Third, a behavioral-level simulator is developed in Matlab calculating the latency and energy that *GraphS* spends on different graph processing tasks. In addition, it has a mapping optimization framework to maximize the performance according to the available resources. ***STT-MRAM:*** We developed a Pinatubo-like [5] accelerator by modifying memory sense amplifiers. Pinatubo is implemented by standard STT-MRAM cell (.cell file in NVSim [19]). ***ReRAM:*** An MPIM-like [6] accelerator with 256 sub-arrays and one buffer sub-array per bank were considered for evaluation. In the computational sub-arrays, for each mat, there are 256×256 ReRAM cells. For evaluation, NVSim simulator [19] was extensively modified to work with Design Compiler [20] to emulate MPIM functionality. Note that the default NVSim's ReRAM cell file (.cell) was adopted for the assessment. ***DRAM:*** We developed an Ambit-like [4] accelerator for graph processing. Ambit implements logic function using capacitor-based majority functions. We accordingly modified CACTI-3DD [21] for evaluation of DRAM's solution. The controllers were synthesized in Design Compiler [20]. ***GPU:*** We used the NVIDIA GTX 1080Ti Pascal GPU. It has 3584 CUDA cores running at 1.5GHz (11TFLOPs peak performance). The energy consumption was measured with NVIDIA's system management interface. We scaled the achieved results by 50% to exclude the energy consumed by cooling, etc. ***Baseline HMC:*** We used a conventional architecture presented in [11] using HMC as main memory without utilizing instruction offloading functionality.

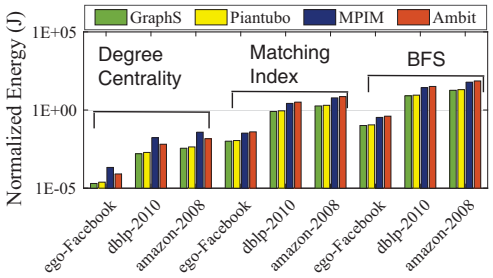

Fig. 7. Normalized log-scaled energy consumption of different accelerators.

B. Overall performance improvement

Fig. 7 shows the normalized energy consumption of under-test accelerators (except GPU and HMC) on various graph processing tasks. As shown, *GraphS* offers the highest energy-efficiency compared to others owning to its low-energy and fully-parallel operations. We observe that *GraphS* achieves on average 4× higher energy-efficiency than that of Ambit accelerator. The main reason here is the energy-efficiency of basic operations in *GraphS*; as discussed earlier, *GraphS* can finish the operations (such as `IML21`) in one-single cycle, however similar operation in Ambit imposes multi-cycle operations avoiding destructive data-overwritten. Fig. 7 shows that *GraphS* solution saves 1.7× and 3.6× energy compared to that of Pinatubo and MPIM solutions. It is worth pointing out that Ambit [4], Pinatubo [5] and MPIM [6] are not capable of implementing parallel one-cycle addition in memory required for different tasks and therefore impose excessive delay and energy consumption to memory chip. To realize such operation in Ambit platform, we consider multi-cycle MG-based implementation [7]. Meanwhile, Pinatubo and MPIM uses multi-cycle XOR logic due to their XOR-friendly architecture. Furthermore, *GraphS* doesn't follow the conventional ReRAM-based crossbar designs to realize PIM, which brings significant energy-efficiency due to eliminating DAC/ADC units. Note that, MPIM and Pinatubo-PCM (not-implemented here) platforms support multi-row operations (up to 256 rows) which can be useful in specific vector processing tasks, however they cannot accelerate most of graph processing tasks which strictly need 2-row operations.

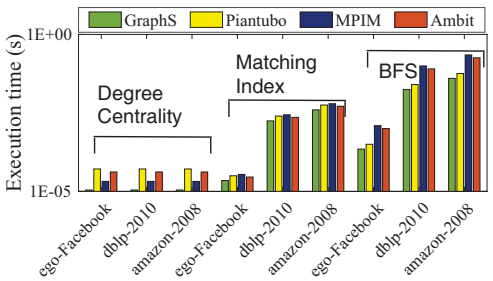

Fig. 8. Normalized log-scaled execution time of different accelerators.

Fig. 8 shows the *GraphS* and other PIMs' execution time on different tasks. It shows that *GraphS* solution is on average 5.1× faster than the DRAM solution (Ambit) and 2.5× faster than STT-MRAM solution (Pinatubo). This is mainly because of ultra-fast and parallel in-memory operations of *GraphS* compared to multi-cycle DRAM/ STT-MRAM operations, specifically for implementing `add` operation. Additionally, we see that *GraphS* is 5.3× faster that ReRAM solution.

We compare the energy saving and speed-up of matching index task running on *GraphS* and baseline HMC normalized with GPU in Fig. 9a and b, respectively, on amazon-2008. We observe that *GraphS* can achieve about 18.4× higher energy-efficiency and 12.6× speed-up compared to GPU while HMC achieves ∼5.4× and 7× improvements, respectively. Fig. 9c gives a quantitative comparison of the state-of-the-art accelerators in energy-efficiency and latency. It can be seen that even though the GPU-based computation can parallelize multiple computations as well, it shows the lowest performance compared to PIM techniques due to memory overhead. Here, Pinatubo implementations (STT-MRAM and ReRAM) show the closest performance to *GraphS*.

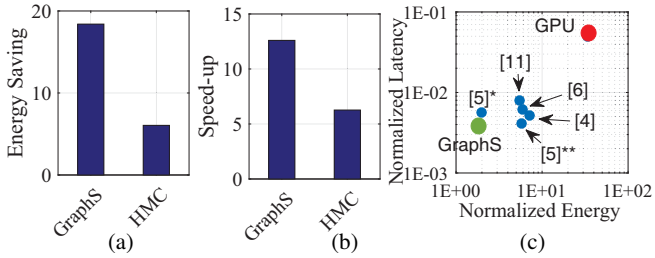

Fig. 9. (a) Energy saving and (b) Speed-up of *GraphS* and baseline HMC normalized to GPU, (c) Latency vs. energy efficiency of different accelerators compared to GPU for matching index operation on amazon-2008 data-set. (*Pinatubo-STT-MRAM **Pinatubo-ReRAM)

C. Memory bottleneck

Fig. 10a depicts the memory bottleneck ratio i.e. the time fraction at which the computation has to wait for data and on-/off-chip data transfer obstructs its performance (memory wall happens) running matching index task on two data-sets. The evaluation is performed according to the peak performance and experimentally extracted results for each platform considering number of memory access in each data-set. We observe that *GraphS* along with other PIM solutions (except MPIM) spend less than ∼ 25% time for memory access and data transfer. Due to unbalanced computation and data movement of MPIM and limitation in number of activated sub-arrays, it shows higher ratio compared to other PIMs. However, GPU accelerator spends more than 90% time waiting for the loading data. The less memory wall ratio can be interpreted as the higher resource utilization ratio for the accelerators which is plotted in Fig. 10b. We observe that *GraphS* can efficiently utilize up to 70% of its computation resources. Overall, PIM solutions can demonstrate high ratio (more than 45% which reconfirms the results reported in Fig. 10a.

V. Conclusion

In this paper, we presented *GraphS* accelerator, which transforms current SOT-MRAM sub-arrays to massively parallel computational units to reduce energy consumption dealing with graph processing tasks and eliminate unnecessary off-chip accesses. The simulation results on three social network data-sets show *GraphS* can roughly achieve 3.6× higher energy-efficiency and 5.3× speed-up over recent ReRAM crossbar and

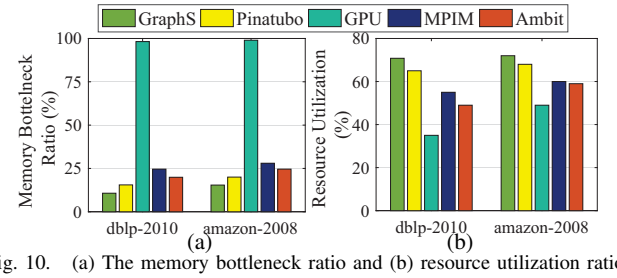

Fig. 10. (a) The memory bottleneck ratio and (b) resource utilization ratio.

4× higher energy-efficiency and 5.1× speed-up over recent processing-in-DRAM acceleration methods.

Acknowledgements

This work is supported in part by the National Science Foundation under Grant No. 1740126, No. 1755761, and Semiconductor Research Corporation nCORE.

References

[1] G. Dai *et al.*, "Graphh: A processing-in-memory architecture for large-scale graph processing," *IEEE TCAD*, 2018.

[2] L. Song *et al.*, "Graphr: Accelerating graph processing using reram," in *HPCA*. IEEE, 2018, pp. 531–543.

[3] S. Li *et al.*, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Micro*. ACM, 2017, pp. 288–301.

[4] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *Micro*. ACM, 2017, pp. 273–287.

[5] S. Li, C. Xu *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*. IEEE, 2016.

[6] M. Imani *et al.*, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *ASP-DAC*. IEEE, 2017.

[7] S. Angizi, Z. He *et al.*, "Design and evaluation of a spintronic in-memory processing platform for non-volatile data encryption," *IEEE TCAD*, 2017.

[8] S. Aga *et al.*, "Compute caches," in *HPCA*. IEEE, 2017, pp. 481–492.

[9] C. Eckert *et al.*, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," *arXiv preprint arXiv:1805.03718*, 2018.

[10] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," *Computer Architecture News*, vol. 43, 2016.

[11] L. Nai *et al.*, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *HPCA*. IEEE, 2017, pp. 457–468.

[12] B. C. Lee *et al.*, "Architecting phase change memory as a scalable dram alternative," in *ACM Computer Architecture News*, vol. 37, 2009.

[13] S. Angizi *et al.*, "Imce: energy-efficient bit-wise in-memory convolution engine for deep neural network," in *Proceedings of the 23rd ASP-DAC*. IEEE Press, 2018, pp. 111–116.

[14] S.-W. Chung *et al.*, "4gbit density stt-mram using perpendicular mtj realized with compact cell structure," in *IEDM*. IEEE, 2016.

[15] K. Navi *et al.*, "A novel low-power full-adder cell with new technique in designing logical gates based on static cmos inverter," *Microelectronics Journal*, vol. 40, pp. 1441–1448, 2009.

[16] (2018) Parallel thread execution isa version 6.1. [Online]. Available: http://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[17] J. Leskovec and J. J. Mcauley, "Learning to discover social circles in ego networks," in *Advances in neural information processing systems*, 2012, pp. 539–547.

[18] (2011) Ncsu eda freepdk45. [Online]. Available: http://www.eda.ncsu.edu/wiki/FreePDK45:Contents

[19] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory," in *Emerging Memory Technologies*. Springer, 2014, pp. 15–50.

[20] S. D. C. P. V. . Synopsys, Inc.

[21] K. Chen *et al.*, "Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory," in *DATE, 2012*. IEEE, 2012, pp. 33–38.