Practical Challenges in Delivering the Promises of Real Processing-in-Memory Machines

Nishil Talati, Ameer Haj Ali, Rotem Ben Hur, Nimrod Wald, Ronny Ronen,
Pierre-Emmanuel Gaillardon*, and Shahar Kvatinsky
Technion – Israel Institute of Technology, Haifa, ISRAEL; *University of Utah, Salt Lake City, UT, USA
{nishil.t, ameerh, rotembenhur, nimrodw}@campus.technion.ac.il, ronny.ronen@gmail.com,
*pierre-emmanuel.gaillardon@utah.edu, shahar@ee.technion.ac.il

Abstract*—Processing-in-Memory (PiM) machines promise to overcome the von Neumann bottleneck in order to further scale performance and energy efficiency of computing systems by reducing the extent of data transfer and offering ample parallelism. In this paper, we take the memristive Memory Processing Unit (mMPU) as a case study of a PiM machine and scrutinize it in practical scenarios. Specifically, we explore the limitations of parallelism and data transfer elimination. We argue that lack of operand locality and arrangement might make data transfer inevitable in the mMPU. We then devise techniques to move data within the mMPU, without transferring it off-chip, and quantify their costs. Additionally, we present electrical parameters that might limit the parallelism offered by the mMPU and evaluate their impact. Using benchmarks from the LGsynth91 suite, their vector extensions, and a few synthetic data-parallel workloads, we show that the internal data transfer results in an increase of up to 1.5× in the execution time, while the parallelism can be limited in some cases to 256 gates, resulting in an increase in execution time by 1.1× to 2×.

***Index Terms*—von Neumann bottleneck, memristors, mMPU.**

I. Introduction

Modern computing systems are designed based on the von Neumann architecture concept, or an improved version of it, which physically separates data processing and data storage entities. Data must therefore be transferred between processor and memory via a bandwidth-limited bus. This data transfer is the biggest bottleneck in current computing systems in terms of both performance and energy [1], and it is commonly known as the *von Neumann bottleneck*. This problem is further aggravated in modern workloads that have little locality and need to compute large amounts of data. Prior research [2] has attempted to mitigate this bottleneck by moving the compute engines near the memory units. However, this solves the problem only to a certain extent since the computation still requires that data be moved out of the memory cells.

This paper focuses on a recently developed real Processing-in-Memory (PiM) architecture called the mMPU – *memristive Memory Processing Unit* [3]–[5]. The key difference between the mMPU and other architectures is that the mMPU does not physically separate processing and memory spaces, and directly employs the memory cells for computation [6] without transferring data out of the memory array. Furthermore, with the ample parallelism offered by the mMPU [3], it can be employed as an efficient vector machine. Hence, mMPU delivers the promise of superior performance and energy efficiency. A careful study of the mMPU in practical scenarios must be performed to evaluate the extent of this promise.

In this paper, we explore the two primary properties of the mMPU – the elimination of data transfer and the enormous parallelism. We show cases when these properties are constrained, present techniques to minimize the effects of those constraints, and evaluate their cost as described below.

- **Data transfer**: the mMPU requires operand locality and alignment to perform computation. Occasionally, data needs to be rearranged to satisfy this requirement. We show how to efficiently perform such data re-ordering and estimate its cost on the execution time.
- **Parallelism**: the mMPU offers plenty of parallelism; however, in practical systems, this parallelism might be limited by electrical parameters. We show the effect of these parameters on the offered parallelism and evaluate the cost of limited parallelism on the execution time.

The rest of the paper is organized as follows. Section II gives background on PiM using the mMPU. In Section III, we describe the necessity of data transfer within the mMPU and propose various mechanisms to minimize it. Section IV discusses the causes and effects of the limited parallelism offered by the mMPU. In Section V, we evaluate the increased workload execution time due to internal data movement and limited parallelism. We conclude the paper in Section VI.

II. mMPU Background

Data in the mMPU is organized in a hierarchical fashion, similar to other memory technologies, where a two-dimensional array of memristive memory cells forms an mMPU-*MAT*. Collections of several such MATs form a *bank*, and multiple banks are gathered to form a *chip* with a common internal bus shared among banks. Within a MAT, depending on cell topology – with or without a selector – it is possible to reach the theoretical minimum in terms of cell area, which is $4F^2$, where F is the feature size (the minimal size of the technology). In accordance with the physical properties of memristive cells, each cell can store one or more data bits in terms of resistance. In this paper, we focus on storing a single bit per cell and performing Boolean logic, so we consider only two discrete resistance levels, referred to as LRS (low resistance state, representing logical 1) and HRS (high resistance state, representing logical 0).

In addition to performing memory operations, the mMPU can also perform logical operations using memristors. Unlike previously proposed PiM architectures that use dedicated processing engines in proximity to memory units [2], the mMPU directly tackles the von Neumann bottleneck by *performing computation using the same memristive devices used as memory cells*, without transferring data outside of the array using a logical family called Memristor Aided loGIC (MAGIC) [3], [7]. Fig. 1a shows a two-input MAGIC NOR gate, which is the basic computation primitive in the mMPU. This primitive employs a parallel combination of input memristors (*i.e.,* in_1 and in_2), and an output memristor (*i.e.,* out) connected anti-serially to inputs. To perform NOR operations, the output memristor is initialized at LRS, after which a MAGIC execution voltage V_0 is applied to the input memristors and the output memristor is grounded. As a result of circuit connections and voltage application, the output memristor produces a result that is the logical NOR of the inputs (*i.e.,* out = in_1 NOR in_2).

Since NOR logic is functionally complete, it is possible to extend NOR operations to compute any arbitrary logical function [5]. Furthermore, the connection pattern among input and output memristors in Fig. 1a shows that MAGIC NOR operations can be performed within the memristive memory crossbar (Fig. 1b) merely by adding a voltage level (*i.e.,* V_0), without any modifications to the crossbar architecture. Since application of V_0 and ground voltages on the BLs extend to all the memristors sharing those BLs, MAGIC NOR operations can be inherently parallellized to realize a vector machine, as shown in Fig. 1b. Transpose memory [3] adds additional flexibility for executing logical functions by adding voltage drivers in the peripheral circuit. Exploiting this flexibility allows logical operations to be performed on operands situated on the same bitline by applying voltages on wordlines.

The mMPU controller [4] is responsible for receiving instructions from the CPU and generating the control signals for the mMPU to perform reads, writes, and logical operations. The key difference between the mMPU controller and on-chip controllers of memories is the arithmetic block, which supports logical instructions in addition to read and write operations. One constraint of processing operands in the mMPU is that their physical addresses should share same wordlines/bitlines (WLs/BLs) since they serve as circuit connections among the inputs and outputs. If two operands that need to be processed are present in different WLs/BLs, they first need to be copied to addresses that share WLs/BLs with other operands, and only then can they be processed. Fig. 1c illustrates this internal data movement using two consecutive MAGIC NOT operations. Note that NOT is a one-input NOR.

III. Data Transfer within the mMPU

Even in the mMPU, data must be moved both within and across different MATs. After explaining why this is the case, we propose mechanisms for internal data transfer within the mMPU without moving data outside the chip and present models to estimate the performance cost of this data transfer.

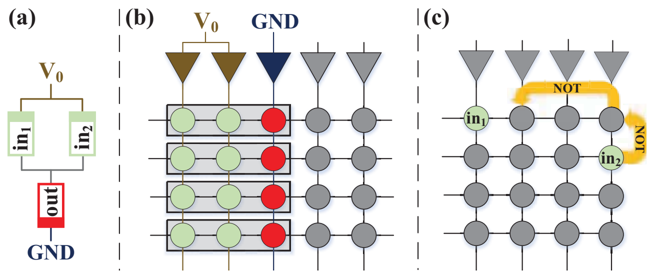


Fig. 1: Computation using the mMPU. (a) Schematic of a two-input MAGIC NOR gate. (b) Parallel execution of MAGIC NOR gates within a MAT. Input memristors are marked in green and output memristors are marked in red. (c) Misalignment of two inputs within the memristive memory array requires moving one of the inputs next to the other, using two consecutive MAGIC NOT operations.

A. Why Data Transfer in the mMPU?

Because the mMPU uses the same cells for processing and memory operations, one might expect the data transfer requirement to be completely eliminated. However, depending on the operand locality and their alignment inside a MAT, this might or might not be possible. For example, consider an in-memory instruction to compute a function with two inputs. Ideally, both input operands would be present in the same MAT, sharing either a common wordline or a bitline (WL/BL), and similarly, the output would also be written to this common data line, making it possible to perform the function in-place without requiring any data movement.

While such operand locality would unleash the full potential of the mMPU, more realistic scenarios might not exhibit such locality. For instance, the input operands could be present in the same MAT yet not share the same WL/BL (Fig. 1c); or the input operands could be present in different MATs or banks. Furthermore, the physical address of the output might be distant from the addresses where the inputs are processed. Hence, depending on the relative locations of the input and output operands, data transfer to contiguous physical addresses might be necessary. This requirement is further exacerbated in the case of vector operations, since large chunks of operands need to be transferred serially. Since transferring data off-chip would incur an enormous overhead, we devise techniques to transfer data only within the mMPU, as will be explained in the next subsection.

B. Techniques for Data Transfer within the mMPU

Depending on the physical addresses of the input and output operands, we categorize the data transfer in following manner:

- **Intra-MAT**, where the operands need to be transferred within a MAT.
- **Intra-bank**, where the operands need to be transferred from one MAT to another within the same bank.
- **Inter-bank**, where the operands need to be transferred from one bank to another within the same chip.

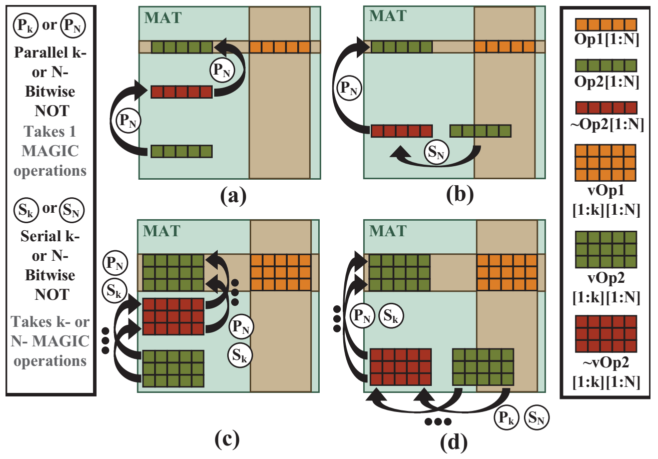


Fig. 2: Data transfer within the MAT when (a) operands do not share any WLs/BLs, and (b) operands partially share WLs/BLs, (c) vector operands do not share any WLs/BLs, and (d) vector operands partially share WLs/BLs.

1) Intra-MAT data transfer: Suppose that in-memory computation is required between two operands, Op1 and Op2, that are present in the same MAT. Assuming that Op2 either does not share data lines with Op1 (Fig. 2a) or only partially shares BLs with Op1 (Fig. 2b), Op2 can be moved using MAGIC NOT operations to align with Op1 within the MAT at a desired location. In the first case, two sets of concurrent MAGIC NOT operations can align Op2 with Op1, whereas in the latter case, Op2 first needs to be moved to a temporary memory location using a serial sequence of MAGIC NOT operations because of the inherent limitation of MAGIC gates not being able to perform multiple operations in the same WL simultaneously [3]. This intermediate result can then be moved to the desired memory location using a single MAGIC NOT cycle.

For vector operations with vectors vOp1 and vOp2 having k-elements each and assuming that all the elements are present in a vector form, the cost of intra-MAT data transfer is even higher since MAGIC NOT operations have to be performed element-wise. As shown in Fig. 2d, if the operand addresses partially overlap, first, the bitwise-MAGIC NOT operations can be performed for all the elements of a vector together, and then, each element can be written to a desired memory location, which takes $(k+N)$–MAGIC NOT operations. When the addresses do not overlap, there are two possible ways to align the vectors, one of which is similar to the case of partial overlapping addresses. The other option is to perform an element-wise copy operation, which takes $2k$–MAGIC NOT operations. Among these, a mechanism with a minimum cost will be selected by the mMPU controller (based on the values of k and N). In total, the latency cost of intra-MAT data alignment for vector operations is

$$C^{\text{intra-MAT}} = \begin{cases} min\{2k, k+N\} \cdot T_{MAGIC}, & \text{no overlap,} \\ (k+N) \cdot T_{MAGIC}, & \text{partial overlap.} \end{cases} \quad (1)$$

The cost of non-vector operand transfer can be found by substituting $k = 1$.

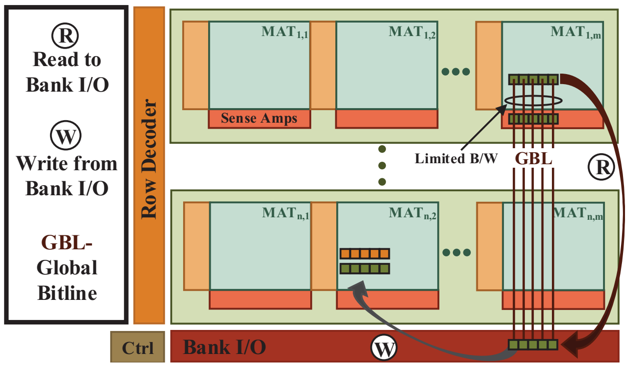


Fig. 3: Illustration of data transfer between different MATs using a read operation to the bank I/O followed by write operation to a desired location in a different MAT.

2) Intra-bank data transfer: When in-memory computation must be performed between operands present in different MATs in the same bank, intra-bank data transfer can be invoked using sense amplifiers, bank I/O, write drivers, and the mMPU controller. Fig. 3 shows the general case, where Op2 must be transferred near Op1 from MAT$_{1,n}$ to MAT$_{m,2}$. To perform an intra-bank data transfer, a sequence of read and write operations is applied, where Op2 is read to the bank I/O, first using the sense amplifier, and then by transferring it to I/O using global bitlines (GBLs). Once at bank I/O, the mMPU controller switches the bank operation from read to write and writes Op2 to the target MAT juxtaposed to Op1. For vector operations, read and write phases must be performed serially for each element of the vector.

Three inevitable micro-architectural considerations for intra-bank data transfers are the turnaround time, the shared row decoder, and the read bandwidth of the sense amplifiers (SAs). First, additional latency is incurred while switching bank I/O circuits from read to write mode and vice versa; these are commonly known as read-to-write T_{RTW} and write-to-read T_{WTR} turnaround latencies. Second, since the row decoder in a bank is shared among different subarrays, only one WL can be activated at a time. As a result, reads and writes cannot be pipelined. Third, because sense amplifiers in non-volatile memories are bulky [8], they might be multiplexed among several bitlines, and the number of bits read simultaneously might be less than a full word. Furthermore, depending on the location of each bit of an operand and the SA multiplexing scheme, reading an operand takes one to multiple read operations. To account for SA multiplexing, we define a variable *B*, which indicates *the number of read cycles required to read an operand*. The cost of intra-bank data alignment for vector operations is therefore

$$C^{\text{intra-bank}} = k \cdot [B \cdot T_{read} + T_{RTW} + T_{write} + T_{WTR}]. \quad (2)$$

The cost of intra-bank data transfer for non-vector operations can be determined by substituting $k = 1$.

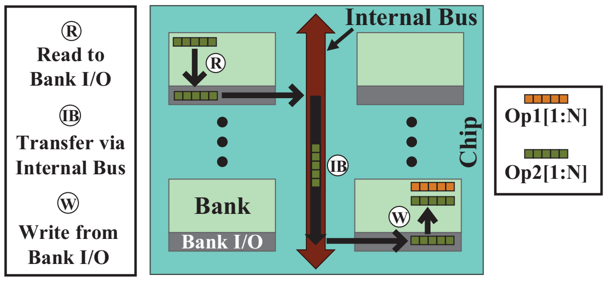


Fig. 4: Illustration of data transfer between different banks within the same chip using the internal bus, a mechanism similar to PSM in RowClone [9].

3) Inter-bank data transfer: When in-memory computation must be performed using operands situated in different banks (Fig. 4), a mechanism similar to Pipelined Serial Mode (PSM) of RowClone [9] can be used to gather operands in the same MAT and perform the computation. Fig. 4 illustrates this transfer mechanism in three steps. In the first step, Op2 is read to its bank I/O via the sense amplifier. Then, in the second step, the operand is transferred to the bank I/O of the destination bank via an internal bus of the chip. Finally, Op2 is written to the desired memory location (adjacent to Op1) in the destination bank. An important timing parameter to consider here is T_{CCD} – column-to-column delay – the delay between two consecutive read/write commands. Hence, in the case of non-vector operations, the mMPU has to wait for at least T_{CCD} to issue a next read/write command. Therefore, the cost of inter-bank data transfer includes three components: reading data to the bank I/O, transfer of data via an internal bus, and writing data to a desired memory location.

If the computation involves vector operands situated in different banks, then different elements of a vector operand can be read, transferred to destination bank using a bandwidth-limited internal bus, and then written to the desired memory location. Since source and destination banks can be activated in read and write modes respectively, inter-bank data transfer can be pipelined among different elements of a vector. While pipelining is desirable for increasing the data transfer throughput, realizing it in non-volatile memories is not intuitive due to the high degree of asymmetry between read and write latencies, and due to process variations. Hence, we do not consider pipelined data transfer in this paper. The cost of inter-bank data alignment for vector operations is

$$\begin{aligned} C^{\text{inter-bank}} = k \cdot [B \cdot T_{read} + max\{T_{CCD}, T_{bus\_transfer}\} \\ + T_{write} + T_{CCD}]. \end{aligned} \quad (3)$$

The cost of inter-bank data transfer for non-vector operations can be found by substituting $k = 1$.

IV. Limited Parallelism

When computing logical functions within the memristive memory, intrinsic parallelism is an attractive performance advantage of the mMPU [3], [5]. This property can be especially beneficial when computing vector operations within a MAT of the mMPU, where the same logical operation is executed throughout all WLs/BLs. In practice, however, this parallelism is limited by many factors, out of which, in this paper, we focus on the parasitic effects of interconnects, memristive device properties, and by the capabilities of the supported periphery circuits (for example, drivers).

Because of the IR drop on the interconnect resistance of WLs/BLs (r_w is the unit resistance between two adjacent WLs/BLs), the voltage at each node decreases as one moves away from the voltage drivers in a MAT. Therefore, in order to support MAGIC NOR operations on all the WLs/BLs, the voltage delivered by the driver after the IR drop should be sufficient for logic operations at the farthest gate. Hence, the parallelism would be limited by the voltage that can be supported by the drivers. The miniaturization of the CMOS technology node would further intensify this effect because higher r_w values mean greater IR drop.

Furthermore, to compute n MAGIC NOR operations in parallel within a MAT, voltage drivers need to push current that is more than the minimum required to program n memristors. This current depends on the data stored in the array in terms of resistance. To support a certain degree of parallelism, the drivers need to support current required in the worst-case data distribution, which is all the memristors programmed as LRS. Hence, the supported parallelism would be limited by the size of drivers in a cost-constrained memristive memory chip, which depends on the interconnect resistance and the LRS value of the memristors.

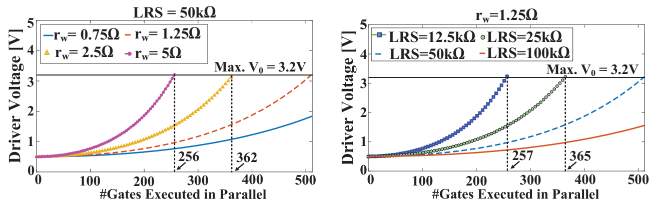
To execute n MAGIC gates in a MAT, we evaluate the driver voltage required ensure that the n^{th} gate (the one farthest from the drivers) receives the minimum V_0 required to execute MAGIC after a voltage drop on the interconnect. Assume HRS>>LRS, the value of MAGIC execution voltage V_0 for each MAGIC gate must be within the range

$$2 \cdot |v_{off}| < V_0 < |v_{on}|, \quad (4)$$

to achieve desired gate functionality without destroying inputs, where v_{off} and v_{on} are the threshold voltages required to program the memristor, and they are equal to logical 0 and 1, respectively. Fig. 5 shows the driver voltage required to attain different amounts of parallelism for different values of r_w and LRS, when $v_{off} = -0.25V$, $v_{on} = 3.2V$, and sneak path currents are eliminated using transistors for each memory cell. It can be observed that the driver voltage needed to be increased tremendously with an increase in r_w and a decrease in LRS. The parallelism of the mMPU can be fully exploited when $r_w \leq 1.25\Omega$ and LRS$\geq 50k\Omega$. Furthermore, even in the worst case with $r_w = 5\Omega$ and LRS$= 12.5k\Omega$, it is possible to execute 256 MAGIC gates in parallel.

V. Evaluation and Results

In this section, we first present our evaluation methodology, then present experimental results that show how data transfer within the chip and limited parallelism affect the execution time of different workloads.



(a) (b)

Fig. 5: Driver voltage required for different amounts of parallelism in a 512 × 512 MAT for different values of (a) interconnect resistance and (b) LRS. Max. V_0, which defines the maximum number of gates that can be executed simultaneously, is determined from (4).

A. *Evaluation Methodology*

We perform a literature survey to extract the latency values of read and write operations for various memristive materials. Instead of selecting a particular device for evaluation, we select the baseline latency values, and then perform a sensitivity study to show the results for a wide range of switching times. To find the latency for MAGIC, we use the VTEAM model [10] to match the desired switching latency and determine the corresponding latency of MAGIC NOR gates using SPICE simulations. The ratio of read, write, and MAGIC latencies is chosen to be $1 : 2.5 : 3.25$ with read latency of 10ns (with B=1), and we use the DDR3-1600 memory interface.

Table I lists the workloads used for evaluation. The first four workloads are the benchmarks synthesized in [5] from the LGsynth91 suite. The next four are the vector extensions of the same benchmarks obtained by replicating the benchmarks as many times as possible in a MAT with 512 WLs/BLs, while keeping the flow of logic execution same as in [5]. Two synthetic workloads that perform vector multiplication and MAC operations are also used. The first one (#9) is a vector fixed point multiplication operation (512 element-wise multiplications). The second workload (#10) is a vector fixed point multiply accumulate operation, where 512 element-wise multiplications are executed, followed by adding the results in the even rows to the odd rows to perform the 256 accumulate operations.

The number of input and output bits are represented as #I_bits and #O_bits. The processing area of a workload represents the number of WL/BL used for in-memory computing (in form of #WL×#BLs). The number of elements in vector operands is represented ad #vElements. The number of MAGIC cycles required to execute each workloads is denoted by #Cycles.

B. *Results and Discussion*

In the following subsections, we show the performance overhead due to internal data transfers and limited parallelism.

1) Impact of Data Transfer: Fig. 6 shows the execution time of workloads for different modes of data transfer normalized to the case of an ideal operand distribution that

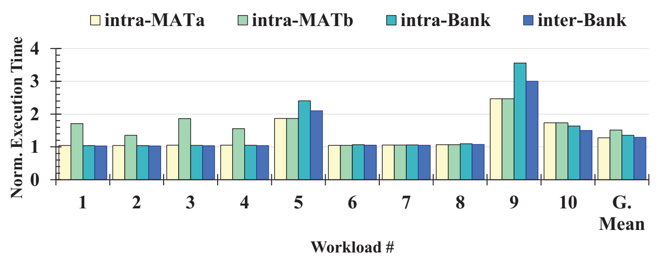


Fig. 6: Execution time for different types of data transfer normalized to the ideal case without any data transfer.

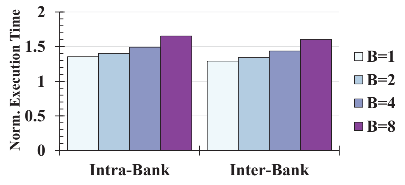


Fig. 7: Average execution time for different values of read bandwidth (B = #read cycles).

requires no data transfer (#cycles in Table I). The results are divided into four categories – intra-MATa (no overlapping operand addresses), intra-MATb (partially overlapping operand addresses), intra-bank, and inter-bank in accordance with the properties of data transfer mechanisms. The average increments in execution time for different workloads in these categories are 27.6%, 51.5%, 35.4%, and 29.9%, respectively. Surprisingly, the cost of data transfer is higher for moving data inside the MAT as compared to moving it outside. This is because MAGIC operations are used for data transfer within the MAT, and their latency is higher than that of read and write operations.

However, moving data outside the MAT consumes about an order of magnitude or more energy than moving it inside the MAT [9]. Furthermore, intra-bank and inter-bank data transfer mechanisms use a serial mode of data transfer among different elements of a vector, whereas intra-MAT data transfer can be parallelized among different MATs in a bank using techniques similar to SALP [11]. Also, Fig. 6 assumes B=1 (B is the #read cycles needed for reading an operand), which is the maximum bandwidth available for reading.

Fig. 7 shows the sensitivity study of the average execution time of workloads as a result of different read bandwidth caused by SA multiplexing and data distribution within the MAT as described in Section III-B. With an increase in B from one to eight, data transfer overhead also increases from 1.35× to 1.65× for intra-bank and from 1.29× to 1.6× for inter-bank modes. Note that overhead of intra-MAT data transfer mechanisms is invariant to changes in B since sense amplifier read operations are not used. With limited read bandwidth, moving data outside the MAT (Fig. 7) is costlier than moving data inside the MAT (Fig. 6), thanks to high bandwidth offered by MAGIC operations. We also performed sensitivity study on

TABLE I: Workload Setup

#	Workload	Source	#I_bits	#O_bits	Processing Area	#vElements	#Cycles
1	cm163a	LGsynth91	16	5	3×61	1	45
2	misex1	LGsynth91	8	7	14×21	1	45
3	parity	LGsynth91	16	1	20×12	1	37
4	x2	LGsynth91	10	7	12×14	1	36
5	Vector cm163a	LGsynth91 + Vector Extension	2720	850	510×61	170	214
6	Vector misex1	LGsynth91 + Vector Extension	288	252	504×21	36	920
7	Vector parity	LGsynth91 + Vector Extension	400	25	500×12	25	709
8	Vector x2	LGsynth91 + Vector Extension	420	294	504×14	42	774
9	Vector Multiplication	Synthetic	8192	4096	512×143	512	354
10	Vector MAC	Synthetic	8192	2048	512×143	256	710

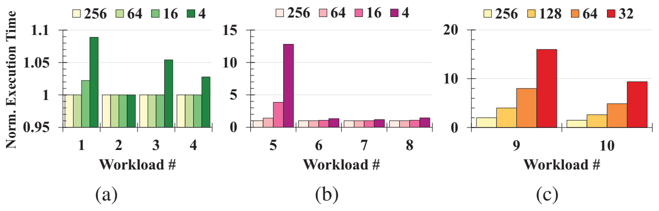


Fig. 8: Execution time for (a) serial, (b) semi-parallel, and (c) heavily parallel workloads. Each bar presents the execution time for a different amount of limited parallelism normalized to full parallelism in a 512×512 MAT.

execution time for a wide range of values of read, write, and MAGIC latencies (not shown due to space limitation), and concluded that it does not change significantly.

C. Impact of Limited Parallelism

Although Fig. 5 depicts that at least 256 gates can be concurrently supported, we further investigate the influence of more severe limitations (for example, due to sneak path currents or different circuit parameters). Fig. 8 shows the execution time of different types of workloads for different degrees of parallelism normalized to the case of full parallelism in a 512×512 memristive array. Fig. 8a and 8b show that except for #5, the execution time increases up to $1.4\times$ (for #8) even with extremely limited parallelism (while executing only four gates per cycle). Whereas, because of the large number of vector elements in #5, a higher degree of parallelism costs a considerable performance penalty in case of limited parallelism. Furthermore, massively parallel workloads (#9 and #10) are affected the most by the limited parallelism as large part of the execution has to be performed serially. For example, #9 performs vector multiplication for all the elements in parallel for the entire execution; hence, reducing parallelism by m–times would increase the execution time proportionally. In #10, however, vector operations are performed with varying degrees of parallelism for 62.8% of the execution time, and the rest is serial. Therefore, the effect of limited parallelism is less severe for #10.

VI. Conclusions

In this paper, we present two practical challenges that could limit the expected benefits of real processing-in-memory (PiM) machines: the data transfer requirement when operand locality is lacking and the limited parallelism. Taking the mMPU as a case study, we present mechanisms to minimize the cost of data movements and preserve the 'in-memory' nature of the computation. We show that these mechanisms cause overhead in execution time of up to $1.5\times$. We also present electrical parameters that might limit the parallelism offered by the mMPU and show how this limited parallelism increases the execution time of different workloads. Evaluating the cost of on-chip data movement and the limitations on parallelism are key steps forward towards the target of demonstrating and evaluating the full benefits from real processing-in-memory systems as compared to standard von Neumann machines.

Acknowledgements

This research was partially supported by the Viterbi Fellowship at the Technion Computer Engineering Center, by the EU ICT COST action IC1401, by the United States - Israel BSF grant no. 2016016, and by the ISF grant no. 1514/17.

References

[1] A. Pedram *et al.*, "Dark Memory and Accelerator-Rich System Optimization in the Dark Silicon Era," *IEEE Design Test*, vol. 34, no. 2, pp. 39–50, April 2017.
[2] R. Balasubramonian and B. Grot, "Near-Data Processing," *IEEE Micro*, vol. 36, no. 1, pp. 4–5, January 2016.
[3] N. Talati *et al.*, "Logic Design Within Memristive Memories Using Memristor-Aided loGIC (MAGIC)," *IEEE Trans. Nanotechnol.*, vol. 15, no. 4, pp. 635–650, July 2016.
[4] R. B. Hur and S. Kvatinsky, "Memristive Memory Processing Unit (MPU) Controller for In-memory Processing," *ICSEE*, 2016.
[5] R. Ben Hur *et al.*, "SIMPLE MAGIC: Synthesis and In-memory MaPping of Logic Execution for Memristor-Aided loGIC," *ICCAD*, 2017.
[6] J. Reuben *et al.*, "Memristive Logic: A Framework for Evaluation and Comparison," *PATMOS*, 2017.
[7] S. Kvatinsky *et al.*, "MAGIC – Memristor-Aided Logic," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, pp. 895–899, Nov 2014.
[8] S. Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," 2016.
[9] V. Seshadri *et al.*, "RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[10] S. Kvatinsky *et al.*, "VTEAM: A General Model for Voltage-Controlled Memristors," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 62, no. 8, pp. 786–790, August 2015.
[11] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," *ISCA*, 2012.