

# PyPIM: Integrating Digital Processing-in-Memory from Microarchitectural Design to Python Tensors

Orian Leitersdorf, Ronny Ronen, and Shahar Kvatinsky

Viterbi Faculty of Electrical and Computer Engineering, Technion – Israel Institute of Technology, Haifa, Israel  
orian@campus.technion.ac.il, ronny.ronen@technion.ac.il, shahar@ee.technion.ac.il

**Abstract**—Digital processing-in-memory (PIM) architectures mitigate the memory wall problem by facilitating parallel bitwise operations directly within the memory. Recent works have demonstrated their algorithmic potential for accelerating data-intensive applications; however, there remains a significant gap in the programming model and microarchitectural design. This is further exacerbated by aspects unique to *memristive* PIM such as partitions and operations across both directions of the memory array. To address this gap, this paper provides an end-to-end architectural integration of digital memristive PIM from a high-level Python library for tensor operations (similar to NumPy and PyTorch) to the low-level microarchitectural design.

We begin by proposing an efficient microarchitecture and instruction set architecture (ISA) that bridge the gap between the low-level control periphery and an abstraction of PIM parallelism. We subsequently propose a PIM development library that converts high-level Python to ISA instructions and a PIM driver that translates ISA instructions into PIM micro-operations. We evaluate PyPIM via a cycle-accurate simulator on a wide variety of benchmarks that both demonstrate the versatility of the Python library and the performance compared to theoretical PIM bounds. Overall, PyPIM drastically simplifies the development of PIM applications and enables the conversion of existing tensor-oriented Python programs to PIM with ease.

## I. INTRODUCTION

As the memory wall continues to limit the performance of data-intensive applications [23], [38], processing-in-memory (PIM) solutions are rapidly emerging to enable logic functionality within the computer memory. The read/write memory interface is supplemented with logic operations where the CPU requests that the memory perform vectored logic on data residing at given addresses, thereby dwarfing data transfer. Whereas early proposals for PIM [12], [37] integrated computation elements *near* memory arrays, emerging PIM proposals exploit the same physical devices for both storage and logic.

Emerging digital PIM architectures (such as DRAM PIM [14], [19], [33], [42], memristive PIM [8], [16], [27], [40], [48], [50] and SRAM PIM [11], [13]) enable bitwise operations within the memory with massive parallelism by designing logic circuits from the same underlying physical devices that construct the memory. For example, DRAM PIM [14], [19], [33], [42] exploits the circuit found in Figure 1(a) to perform majority logic between the capacitors. The circuit from Figure 1(a) is found in every column<sup>1</sup> of the DRAM subarray, thereby enabling parallel execution of the same logic gate amongst all aligned columns in the subarray and all subarrays

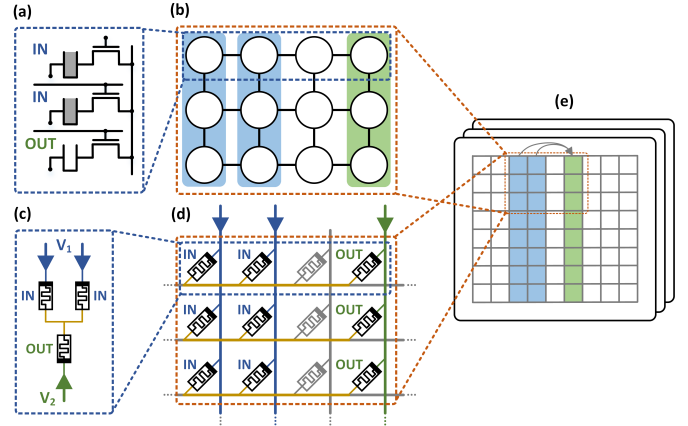


Fig. 1. (a) Majority logic [14], [19], [33], [42] within (b) all rows of a DRAM subarray. (c) Stateful logic [8], [16], [27] between memristors within (d) all rows of a crossbar array. Both support (e), an abstract model enabling arbitrary bitwise operations on columns. The figure is adapted from AritPIM [29].

in the memory. Similarly, memristive [10] memories enable logic functionality according to the circuit in Figure 1(c) [8], [16], [27], which can then similarly be repeated across all rows of a crossbar array for massive throughput [48], [50]. Unlike DRAM PIM, memristive PIM also simultaneously supports parallel operations across columns due to the symmetry of memristor arrays; further, the parallelism of memristive PIM may be increased through partitions [3], [16], [29], [31], [34]. Therefore, in this paper, we propose a framework that supports *partition-enabled memristive PIM* as the most complex case for both the low-level microarchitecture and the high-level programming model. We further consider an inter-array communication framework that consists of an H-tree hierarchy to enable distributed communication among memristor arrays.

The algorithmic challenge in the integration of digital PIM arises from the efficient exploitation of the massive bitwise parallelism. Since the gates can only be performed in parallel when they are aligned, we strive to design algorithms that maximize the gate alignment throughout the computation. The first algorithmic step involves constructing high-throughput *vectorized arithmetic* from the underlying basic bitwise operations. AritPIM [29] recently proposed a suite of high-throughput arithmetic operations for both fixed-point and floating-point numbers that is based on the *element-parallel* approach for performing vectored arithmetic in parallel across all rows of the memory. The next step expresses applications such as

<sup>1</sup>The sub-array is illustrated transposed in Figure 1(b).

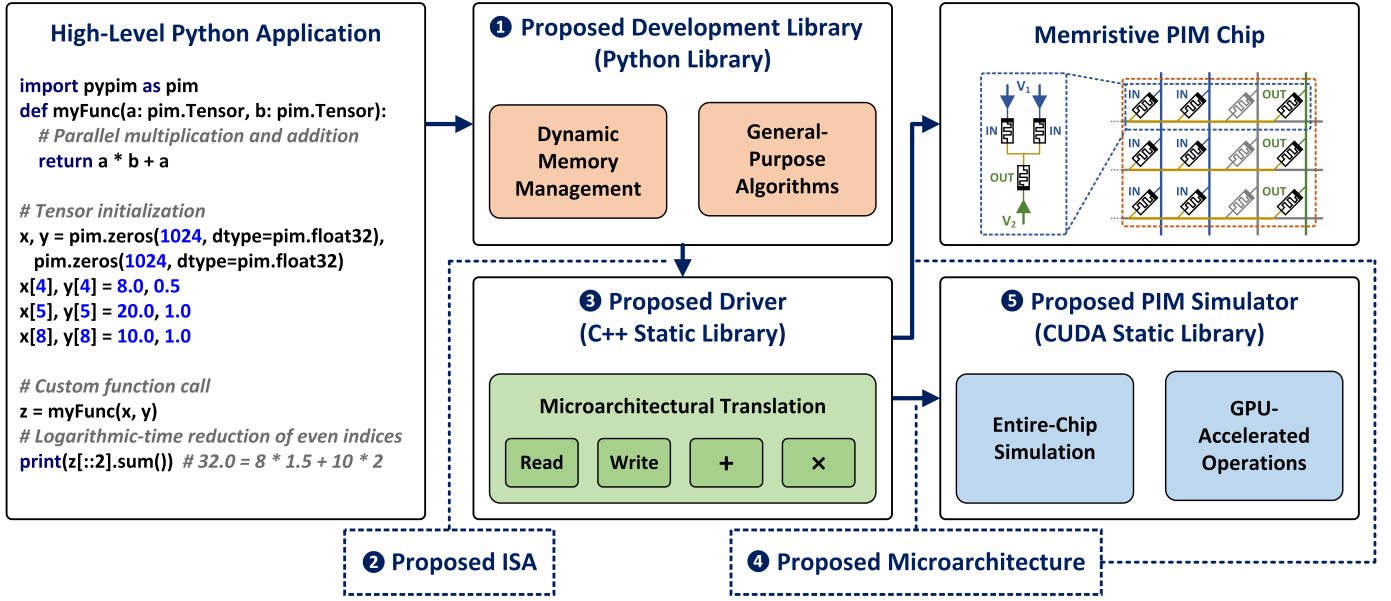


Fig. 2. End-to-end integration from high-level Python to the proposed microarchitecture (arrows indicate runtime dependencies), thereby enabling the development and debugging of PIM applications. The Python library utilizes syntax similar to NumPy [20] for vector arithmetic (e.g.,  $a * b + a$ ), read/write operations (e.g.,  $x[4] = 8.0$ ), indexing (e.g.,  $z[::2]$  selects all even indices), and general-purpose routines (e.g.,  $.sum()$  for aggregation).

matrix multiplication [30] and FFT [28] in terms of these vectored arithmetic operations, while also manually managing the alignment of the vectors in the memory.

While significant effort has been invested in the algorithmic development of digital PIM, there have been only a few works on the underlying microarchitecture, controller design, and programming model [13], [19], [48], [50], [53]. This paper aims to provide the end-to-end architectural integration for memristive PIM that intends to bridge the gap between high-level algorithmic theory and low-level logic design. Figure 2 is an overview of this goal, providing a familiar Python development environment for tensor operations that is automatically translated into the parallel low-level operations that adhere to the proposed microarchitecture. We further propose a high-performance GPU-accelerated digital PIM simulator that models the proposed microarchitecture as a drop-in replacement for the physical chip, thereby enabling a ready-to-use platform for developing and debugging of PIM algorithms. Overall, PyPIM is the culmination of the following five components:

- **Development Library (Section V-A) ①:** We propose a PIM development library that enables high-level Python code (seen on the left of Figure 2) to automatically exploit PIM parallelism. The library includes PIM-optimized dynamic memory management, intra-array and inter-array PIM algorithms, and Python operator overloading techniques that enable PIM development with ease. Furthermore, the library enables the design of new PIM routines (e.g., `myFunc` in Figure 2) using traditional Python semantics, and provides general-purpose routines such as vector reduction (summation) in logarithmic time [41] using `.sum()` and efficient distributed communication for tensor alignment and shift operations.

- **Instruction Set Architecture (Section IV) ②:** We propose a theoretical model for digital memristive PIM that is based on a model of warps and threads (crossbars and rows, respectively). The goal is to abstract the implementation details (e.g., the supported logic gates) while maintaining the massive throughput and flexibility of PIM. This both simplifies algorithmic development and improves generalization across PIM architectures.
- **Host Driver (Section V-B) ③:** We propose an efficient driver that translates macro-instructions (instruction-set architecture) into micro-operations (micro-architecture). The driver includes elementary arithmetic routines (addition, subtraction, multiplication, and division) on fixed-point and floating-point numbers adapted from Arith-PIM [29], and several new miscellaneous routines such as comparison and multiplexing to complement the suite of supported instructions. While previous works assumed that this translation is performed via a dedicated hardware controller, we designed an efficient host program that supports the PIM parallelism. This provides greater flexibility than a hardware controller as the driver may be modified in the future for additional functionality.
- **Microarchitecture (Section III) ④:** We propose a microarchitecture for digital memristive PIM that expands the traditional read/write interface to support efficient operation decoding for partitions, flexible crossbar addressing, flexible row isolation, and hierarchical H-tree inter-crossbar communication. This is accomplished first through a variety of novel techniques that express the wide range of possible partition operations with a minimal number of bits, and then through the standardization of the operation format for the micro-operations.

- **GPU-Accelerated Simulator (Section VI) ☉:** We develop a bit-level digital PIM simulator that interfaces with the host driver as a replacement for the physical digital PIM chip. The simulator is itself GPU accelerated to reduce simulation time and enable the simulation of large-scale applications. Overall, this enables the execution, debugging, and profiling of PIM applications with ease.

This paper is organized as follows. We begin in Section II with general background on memristive digital PIM and a summary of the related architectural works. We then continue in Section III with the proposed microarchitecture and in Section IV with the proposed instruction set architecture (ISA). In Section V, we propose the development and the driver libraries that enable digital PIM algorithm development with significant ease, and in Section VI we evaluate the libraries using the proposed GPU-accelerated simulator on a wide variety of benchmarks. Section VII discusses miscellaneous considerations, and Section VIII concludes this paper.

## II. BACKGROUND

This section begins by providing background on digital memristive PIM architectures from both the circuit and the theoretical algorithmic perspectives, and then continues with a review of previous works on the circuit and architecture considerations from various PIM types.

### A. Digital Memristive PIM

Memristive PIM architectures [9], [16], [24], [32], [41], [47], [48], [50] utilize the emerging nonvolatile memristor device [10], [43] towards a dense computer memory that is inherently capable of both information storage and logic. While memristors may store multiple bits and serve as efficient analog matrix multiplication accelerators, in this work we use memristors for digital computation to achieve improved accuracy and generality [45]. This is accomplished via the stateful logic [40] technique which provides the foundation for high-throughput bitwise operations.

The memristor is a nonvolatile two-terminal resistive device that is inherently capable of both information storage and digital logic. Memristors possess a variable resistance that is modified through a strong current – typically, a current in one direction may increase the resistance, whereas a current in the opposite direction decreases it. Therefore, memristors support binary information storage through their resistance, where the resistance domain is split into a binary classification (high resistance for logical zero and low resistance for logical one). Writing is performed with a relatively high current, whereas reading is performed by measuring the current from a low voltage. Furthermore, stateful logic [8], [16], [27], [40] enables logic operations between memristors in the resistance domain by applying fixed voltages. Consider the circuit in Figure 3(a) assuming  $V_1 \gg 0V$ ,  $V_2 = 0V$ , and that the output memristor is initialized to low resistance: only if the resistance of at least one of the input memristors is low (logical one), then a relatively high current will flow through the output memristor and switch it to high resistance (logical zero). That is, a

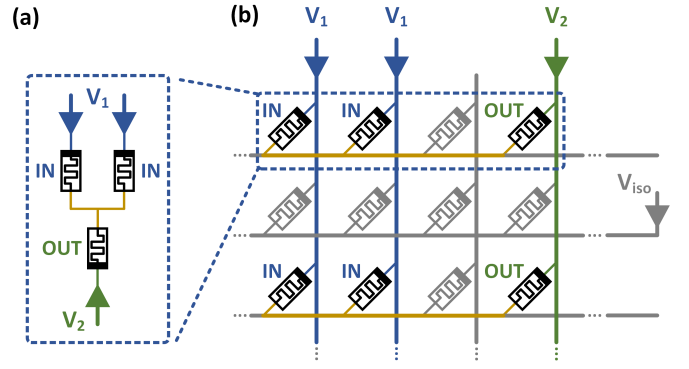


Fig. 3. (a) Stateful logic [40] in the resistance domain between three memristors. (b) Parallel stateful logic in a crossbar array by applying  $V_1$  and  $V_2$  across bitlines while skipping a row, e.g., using  $V_{iso}$ .

logical NOR is performed between the resistance states of the input memristors, with the result being stored in the output memristor [27]. This logic technique has been extensively studied from the circuit perspective and already has several experimental demonstrations across different memristive devices [8], [21], [22], [44], [54].

Memristive crossbar arrays enable high-density memories that also support parallel stateful logic operations. Such crossbar arrays are typically formed from a grid of, e.g.,  $1024 \times 1024$  memristors connected to vertical bitlines and horizontal wordlines, as seen in Figure 3(b). Write operations are performed by grounding a specific wordline and then applying a high voltage across the bitline(s) of the corresponding memristor(s). Read operations are performed similarly, albeit with a lower applied voltage and with a current sense amplifier connected to the bitline(s). Lastly, logic operations are performed by applying the fixed voltages  $V_1$  and  $V_2$  on three arbitrary bitlines, leading to the circuit seen in Figure 3(a) forming in all rows of the crossbar *in parallel*. By broadcasting the same instruction to (up to) all of the crossbars in the overall memory, this parallelism may be extended to massive throughput for aligned bitwise operations. It is also possible to *skip* (deselect) rows in stateful logic by, for example, applying an isolation voltage  $V_{iso}$  to the corresponding wordline.

### B. Element-Parallel Arithmetic

The element-parallel arithmetic technique extends the bitwise parallelism towards high-throughput vectored arithmetic. Consider a crossbar of dimensions  $h \times w$  containing two  $h$ -dimensional  $N$ -bit vectors  $\vec{u}, \vec{v}$ , where each row contains a single element from each vector. This technique performs a vectored operation (e.g., vector addition) on  $\vec{u}, \vec{v}$  in parallel across all rows, in one of the following configurations:

- The *bit-serial* element-parallel approach [7], [17], [29], [48], illustrated in Figure 4(a), accomplishes this task by expressing the arithmetic operation as a serial sequence of logic gates which are executed in parallel across all rows. For example,  $N$ -bit addition is performed by first constructing a full-adder from 9 serial NOR gates, and

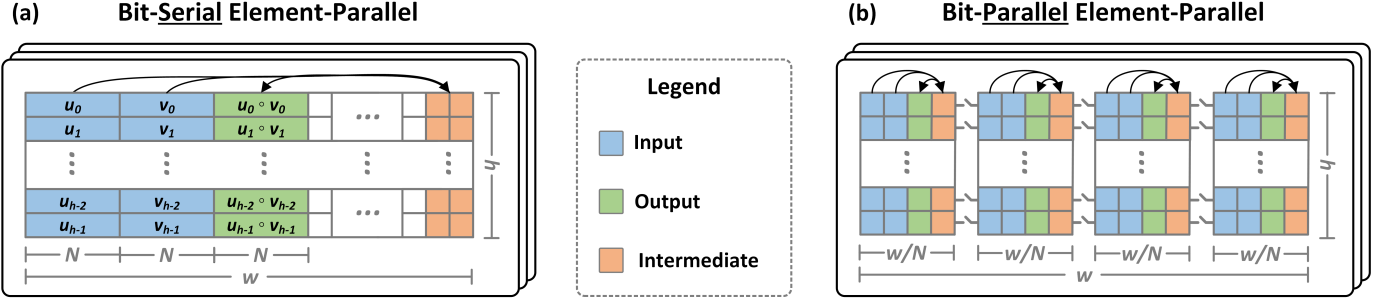


Fig. 4. (a) Bit-serial element-parallel arithmetic constructs vectored arithmetic from a serial sequence of logic gates that is performed in parallel across all rows (one gate per row in every cycle). Conversely, (b) the bit-parallel element-parallel approach stores the vectors in a *strided* format across  $N$  partitions (each bit position in a different partition) and performs up to  $N$  gates per row per cycle. Figure adapted from AritPIM [29].

then implementing ripple-carry addition in  $9N$  cycles. While this approach possesses long latency, it also provides high throughput from the concurrency across rows.

- Conversely, the *bit-parallel* element-parallel [29], [31], [34] approach strives to provide both low latency and higher throughput by introducing dynamically-connected (transistor-based) *partitions* that enable multiple gates per row per cycle. For example, with  $N$  partitions, the vectors can be stored in a bit-strided format, as shown in Figure 4(b). When the partitions are disconnected, the maximal parallelism of  $N$  gates per row is possible in every cycle; the partitions may also be connected to allow gates between different partitions, enabling information transfer between them. This may accelerate arithmetic through algorithms such as carry lookahead [29] by enabling up to  $N$  concurrent gates for each arithmetic operation, e.g., reducing  $N$ -bit multiplication latency from  $O(N^2)$  to  $O(N \log(N))$  ( $14\times$  improvement for  $N = 32$  [29]). Essentially, this reduces the latency from the total gate count towards the length of the critical path.

Such high-throughput arithmetic is exploited towards the acceleration of data-intensive applications. These applications can be broadly split into intra-crossbar and inter-crossbar. Intra-crossbar applications focus on extending the vector parallelism within a single crossbar to tackle more complex problems, such as matrix operations [18], [24], [30] and Fast-Fourier-Transform (FFT) [28], where the application is primarily expressed as a sequence of vectored arithmetic operations and the data layout within the crossbar is carefully managed to guarantee data alignment. Conversely, inter-crossbar applications utilize the entire memory towards a larger application, such as neural networks [24], [25], graph operations [32], DNA sequencing [15], [26], and databases [39]. Inter-crossbar applications typically consist of intra-crossbar routines performed in parallel across all crossbars, supplemented with data transfer (i.e., using read and write circuitry) among crossbars via distributed communication frameworks such as H-trees.

### C. Related Work

Previous architectural research into digital memristive PIM primarily focused on designing and evaluating the peripheral

circuitry rather than the programming model. Talati et al. [46], [48] and Wald et al. [52] thoroughly investigated the circuit and peripheral considerations, including the effect of non-ideal wires on the logical correctness and the design of the peripheral circuits that apply the stateful logic voltages. Further, RACER [50] continued this work, evaluating additional electrical considerations and designing peripheral circuits for an entire memristive memory architecture. Therefore, in this paper, we assume the electrical and peripheral correctness from these previous works and we instead focus on the microarchitecture and its extension to the high-level tensor-based Python programming interface.

There has also been initial development of the extension to the programming interface for in-DRAM [19], non-partition memristive [39], [49], [50], and in-SRAM [13], [53] architectures. Previously proposed ISAs [13], [19], [50] have abstracted bit-serial arithmetic according to a predefined set of instructions performed in parallel across several rows. Duality Cache [13] has further detailed the potential parallelism through a model similar to that of warps and threads in CUDA, and has provided a conversion from NVIDIA PTX assembly to Duality Cache instructions. Yet, these previous works do not support partition-based computation, flexible crossbar/row isolation, and computation across both directions of the array. Further, in this paper, we propose additional aspects such as (1) the dynamic memory management and mapping of aligned tensors, (2) the abstraction of inter-crossbar communication through general-purpose routines such as logarithmic reduction and tensor masking, and (3) the familiar NumPy [20] syntax and algorithms provided by the Python library. Lastly, while previous works have designed on-chip controllers [19], [39], [50], we propose a host driver that both provides greater flexibility (as the driver may be updated without replacing the hardware) and is not a bottleneck to PIM performance.

## III. MICROARCHITECTURE

This section details the proposed microarchitecture for digital memristive PIM. The microarchitecture supports efficient operation encoding for partitions, flexible range-based crossbar addressing, flexible range-based row isolation, and H-tree inter-crossbar communication. The micro-operations in this microarchitecture (referred to as “operations” for simplicity)



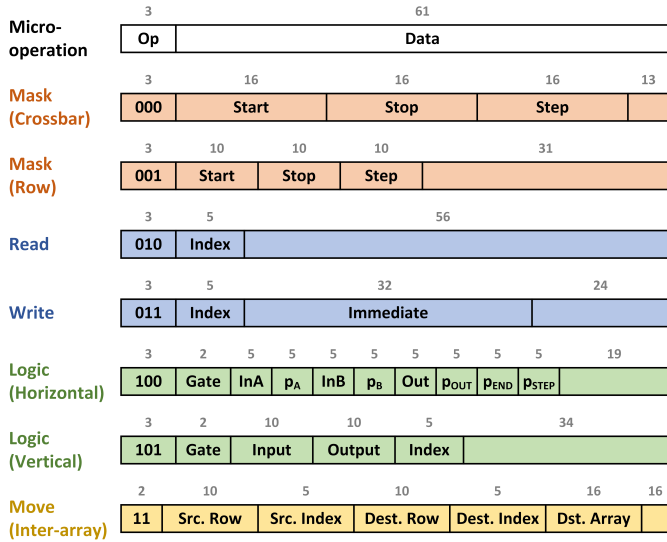


Fig. 5. An overview of the different proposed micro-operation types.

are generated by the host driver proposed in Section V and are broadcasted to all crossbars. As the operations directly translate into the voltages for the crossbar periphery, then the on-chip controller only needs to buffer the operations and broadcast them to the crossbars. We begin with an overview of the microarchitecture and operations in Section III-A and Figure 5, and then continue by detailing each operation.

#### A. Overview

The proposed microarchitecture supports four different operation types that enable both memory and logic functionality. For simplicity, we present here the case of  $h \times w = 1024 \times 1024$  crossbar size with  $N = 32$  partitions comprising an 8GB memory (64k crossbars) [29], [41] that supports NOT and NOR operations in the horizontal direction and NOT operations in the vertical direction. Regardless, the libraries provided in Section V can be configured according to different parameters and gates, there are sufficient unused bits in the format for larger memories, and the proposed mechanisms can be generalized to the case where the number of partitions differs from  $N$  (the size of a word in the architecture). The microarchitecture interface consists of 64-bit operations sent from the host driver, with an optional  $N$ -bit response for read operations. The supported operation types are:

- 1) *Mask*: These set the per-crossbar and per-row masks, indicating which rows are active in the next operations.
- 2) *Read/Write*: Standard read/write operations to the memory with  $N$ -bit granularity. The target crossbar(s) and row(s) are specified in preceding *mask* operations, and then the intra-row index is specified in the operation.
- 3) *Logic*: These operations communicate a logic operation and are split into *horizontal* operations (as seen in Figure 3) that encode partition operations and *vertical* operations that essentially transfer data between two rows (e.g., using a NOT gate in the transposed direction).

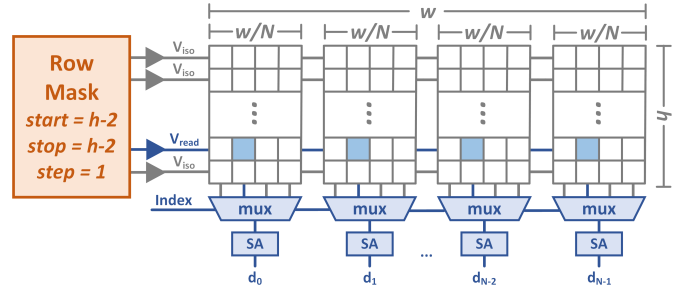


Fig. 6. An overview of the strided data access of the reading mechanism [46].

- 4) *Move*: These operations communicate a parallel *distributed* inter-array data movement among the arrays in an H-tree hierarchical structure.

#### B. Mask Operations

These operations select the rows of the memory that will be activated in the following read/write and logic operations. While the maximal PIM throughput is attained when all crossbars and all rows participate in the computation, it may be required to only operate on selected crossbars or rows (e.g., using isolation voltages, see Section II-A) to either avoid corrupting unselected data or reduce energy consumption. We support a range-based pattern for these masks, defined according to *start*, *stop*, and *step* values as  $\{start, start + step, start + 2 \cdot step, \dots, stop\}$  (where they must satisfy that *step* divides *stop* - *start*). This pattern enables the flexibility required by previous PIM applications works and requires a small representation size.

The crossbar mask is implemented with every crossbar storing a single volatile bit representing whether that crossbar is currently activated. Whenever a crossbar mask operation is broadcasted, the peripheral circuitry of every crossbar updates the stored activation bit accordingly. For the following non-mask operations, the stored activation bit acts as an enable bit for the entire operation (i.e., the operation is not performed in the given crossbar if the stored activation bit is false).

The row mask is implemented by the crossbars storing the *start*, *stop*, and *step* values and utilizing them in non-mask operations. The row mask is updated by a row mask operation that applies to all crossbars and provides the updated *start*, *stop*, and *step* values. During read/write and horizontal logic operations, the row mask is expanded into a binary vector of length  $h$  representing the activated rows and is then used as the enable bits for the desired operation (for example, whether to apply  $V_{iso}$  across rows for stateful logic operations).

#### C. Read/Write Operations

The read/write operations enable standard memory access in addition to the PIM functionality. The operations access the data in  $N$ -bit granularity in a strided memory format, where preceding mask operations first select the desired crossbar(s) and the row(s), and then the read/write operation specifies the intra-row index. Read operations are performed following mask operations that select a single row in a single crossbar

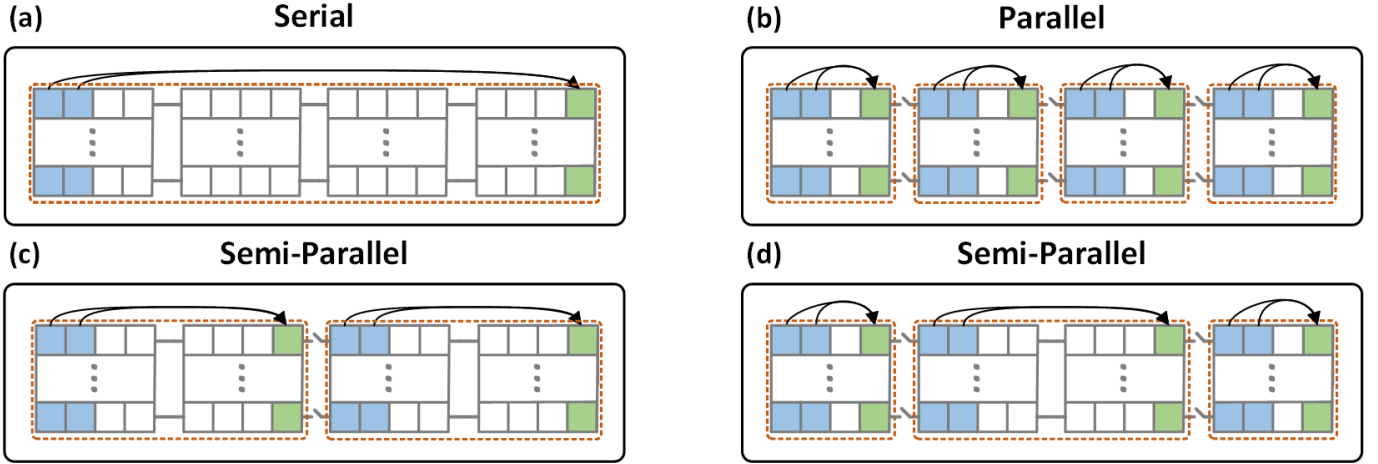


Fig. 7. Overview of the different types of partition-based parallelism: (a) serial, (b) parallel, and (c,d) semi-parallel. The dynamic section division is shown in dashed orange, inputs are blue, and outputs are green.

and further provide a  $\log(w/N)$ -bit index specifying the intra-row strided address. Figure 6 exemplifies this case for  $N = 4$  where row  $h - 1$  is selected at index 2. The strided memory access is due to the sense amplifiers being shared amongst several consecutive bitlines to reduce area [46], [50], yet this also coincides with the strided memory format utilized in bit-parallel element-parallel computation as there is a single multiplexer for each partition. Write operations are performed similar to Figure 6, yet the mask may select multiple crossbars and rows to enable parallel write operations for the same data.

#### D. Logic Operations (Horizontal)

We propose an operation format for encoding partition operations. We begin in Section III-D1 with a review of the different forms of partition parallelism (serial, semi-parallel, and parallel) from the perspective of the computational model [3], [29], [31], [34], continue in Section III-D2 by proposing the *half-gates* technique which supports this parallelism with standard crossbar periphery, and then conclude in Section III-D3 with the overall proposed operation format that simultaneously provides the flexibility required by previous algorithmic works and a relatively-small encoding size.

1) *Partition Parallelism*: Partitions enable a unique parallelism that may be exploited for efficient arithmetic techniques. Consider inserting  $N - 1$  transistors at fixed locations into each row of an  $h \times w$  crossbar, as illustrated in Figure 7. The transistors dynamically isolate different parts of each row to enable concurrent execution, essentially dynamically dividing the crossbar partitions into *sections* (dashed orange) such that each section may perform a parallel stateful logic operation. Initial works [3], [16], [34] utilized partitions in a binary fashion: either the entire crossbar is one section (serial – see Figure 7(a)) or each partition is a section (parallel – see Figure 7(b)). Recent works demonstrated the potential of semi-parallelism, e.g., a further  $4\times$  improvement for multiplication [29], [31] compared to the previous binary solution [34]. We define these parallelism forms, presenting

a trade-off between parallelism (gates per cycle per row) and flexibility (inputs and outputs from different partitions to facilitate communication between bit positions):

- *Serial* (Figure 7(a)): When the transistors are *all conducting*, the crossbar is equivalent to one without partitions. Thus, only a single gate is executed per row per cycle.
- *Parallel* (Figure 7(b)): When the transistors are *all not conducting*, then  $N$  gates may operate concurrently as part of an operation, *one gate within each partition*, or  $N$  gates per row per cycle.
- *Semi-Parallel* (Figures 7(c,d)): When only *some* transistors are *conducting*, then multiple gates may operate concurrently, *between partitions*. Essentially, the sections that define the concurrent gates must not intersect.

2) *Half-Gates Technique*: This section provides the core technique utilized in PyPIM to support the abstract parallelism provided by partitions using standard crossbar periphery components. The traditional operation format for horizontal logic in non-partition crossbars includes three-column indices that specify the input and output bitlines. Further, the operation also specifies the gate type to be performed; similar to previous works [29], [48], [50], without loss of generality, we assume four operations of  $\{INIT0, INIT1, NOT, NOR\}$ , where  $INITx$  is a constant logic gate without inputs that sets the output to  $x$  (similar to a write operation). The periphery for such a crossbar consists of a column decoder that receives the indices  $InA$ ,  $InB$ , and  $Out$ , and applies  $V_1$  on bitlines  $InA/InB$  and  $V_2$  on bitline  $Out$ , see Figure 8(a); the voltages  $V_1$  and  $V_2$  are chosen according to the gate. The column decoder consists of three decoders, each of which receives a column index and outputs either  $V_1$  or  $V_2$  [6], [40], [46].

Our proposed approach is based on a novel technique of *half-gates*: we utilize a single-column decoder per partition, and introduce per-partition opcodes that are exploited towards partition parallelism, as shown in Figure 8(b). We describe the basic idea through the following example: to support a stateful logic gate where inputs are in partition  $p_a$  and outputs

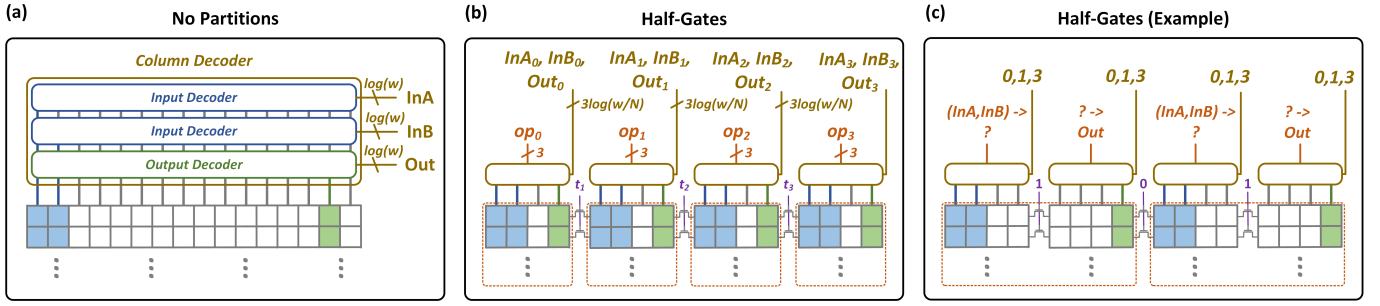


Fig. 8. (a) The stateful logic periphery for a baseline crossbar with no partitions [46]. (b) The periphery as proposed by the *half-gates* technique, with (c) an example for semi-parallelism corresponding to Figure 7(c).

TABLE I  
PER-PARTITION OPCODES IN THE HALF-GATE TECHNIQUE.

Index	Opcode	Index	Opcode
000	-	100	$(InA, ?) \rightarrow ?$
001	$? \rightarrow Out$	101	$(InA, ?) \rightarrow Out$
010	$(?, InB) \rightarrow ?$	110	$(InA, InB) \rightarrow ?$
011	$(?, InB) \rightarrow Out$	111	$(InA, InB) \rightarrow Out$

are in partition  $p_b$  (both in the same section), (1) the column decoder of  $p_a$  applies only the input voltages without applying the output voltages, and (2) the column decoder of  $p_b$  applies only the output voltages without applying the input voltages. Essentially,  $p_a$  “trusts” that a different partition will apply output voltages, and  $p_b$  “trusts” that a different partition will apply input voltages. While each gate on its own is not valid (*half-gate*), their combination is valid. Table I details the various possible opcodes for each column decoder, where “?” represents not applying voltages for that part of the gate and “-” represents not applying voltages at all (e.g., for partitions in between  $p_a$  and  $p_b$ ). An example is shown in Figure 8(c) for the operation from Figure 7(c). The opcode decoding utilizes the first two bits as the enable bits for the input decoders, and the last bit as the enable bit for the output decoder.

3) *Partition Model*: While the model proposed in the previous section (and highlighted in Figure 8(b)) can support the full flexibility of semi-parallelism from Section III-D1, this would lead to a massive operation encoding due to the vast number of partition operations possible. Therefore, this section proposes three restrictions on the model that enable a compact operation format that still supports the previous algorithmic PIM works as they already adhere to these patterns.

The first restriction requires identical *intra-partition* indices; that is,  $InA_0 = InA_1 = \dots$ ,  $InB_0 = InB_1 = \dots$ ,  $Out_0 = Out_1 = \dots$ . The example in Figure 8(c) already satisfies this requirement as the same  $InA, InB, Out = 0, 1, 3$  are inputted to all of the column decoders. Notice that, as in the case of Figure 8(c), some of the operands are not utilized by the partitions (e.g., a partition with opcode “ $(InA, InB) \rightarrow ?$ ” will not use  $Out$ ), yet this does not affect the correctness.

The second restriction is that the opcodes repeat periodically. Consider the opcodes representing the leftmost gate (e.g., the opcodes of “ $(InA, InB) \rightarrow ?$ ” and “ $? \rightarrow Out$ ” in

the first and second partitions of Figure 8(c), respectively). We require that the opcodes for the remaining concurrent gates in the operation repeat; e.g., the opcodes in the third and fourth partitions in Figure 8 are a repetition of the opcodes in the first and second partitions. The operation encoding includes (1) the partition indices  $p_A, p_B, p_{OUT}$  of the two inputs and the output of the leftmost gate (where  $p_A \leq p_B$ ), and (2) the periodicity represented by the index of the partition containing the output of the last gate  $p_{END}$  and the step size  $p_{STEP}$ .

The third restriction aims to deduce the transistor selects from the opcodes. The partition model as defined above enabled some flexibility in the selection of the transistor selects for the same set of gates. For example, if there are two concurrent gates, one from partition 0 to partition 3 and another from partition 8 to partition 11, then any one of the transistors between partition 3 and partition 8 can be set to non-conducting for the operation to be valid. For the case of  $p_A \leq p_{OUT}$  ( $p_A > p_{OUT}$  is similar), we restrict the transistor selects to adhere to the following pattern: a transistor is set to non-conducting only if the partition to its left has opcode  $* \rightarrow Out$  (where  $*$  reflects “don’t care”) or the partition to its right has opcode  $(InA, *) \rightarrow *$ . This restriction enables the generation of the transistor selects from other existing fields.

Overall, we find that this operation format requires  $2 + 3 \cdot \log(w) + 2 \cdot \log(N) = 42$  bits in total (gate type, input/output indices, and opcode pattern), only a  $1.31\times$  increase over that of a crossbar without partitions. Figure 5 details this format, providing 19 unused bits out of the available 64 bits. We find that this restricted partition model still supports the full flexibility of the previous algorithmic works as it supports the underlying fundamental routines utilized in those works. Previous works that have utilized the high flexibility of semi-parallelism have first designed useful general-purpose partition techniques such as broadcast and reduction operations [29], [31], and have then utilized those routines to tackle larger problems such as parallel prefix carry-lookahead addition [29]. As the operations utilized in the general-purpose partition techniques adhere to the minimal partition model, we find that this model may also support the more complex arithmetic algorithms. We demonstrate this in Section V by implementing the AritPIM [29] suite using the proposed microarchitecture.

### E. Logic Operations (Vertical)

Stateful logic also supports operations in the transpose direction when the voltages  $V_1, V_2$  are applied on the wordlines rather than the bitlines [48]. These operations are primarily utilized to transfer data *between different rows of the same array*, such as using two consecutive NOT gates [41], as arithmetic in the transpose direction is not possible *when  $N$ -bit numbers are stored across  $N$  horizontal cells*. Therefore, we support only the  $\{INIT0, INIT1, NOT\}$  set of gates for vertical stateful logic operations. The operation format includes the input and output rows for the vertical gate, as well as the *Index* field that represents the column mask. That is, the vertical logic operation is applied to the columns that are at an intra-partition index equal to *Index*, similar to the access pattern of read/write operations.

### F. Move Operations (Inter-Array)

We propose an inter-array communication framework based on a hierarchical H-tree structure connecting the crossbars. Previous works [24], [39], [50] have required data movement within and between crossbar arrays for the acceleration of data-intensive applications. Notice that intra-crossbar data transfer (see Section III-E) is supported with massive parallelism since it may occur in parallel across all crossbar arrays, yet achieving inter-crossbar data transfer with the above operations (i.e., perform a read operation from one crossbar and then a write operation to the other crossbar) leads to entirely serial data movement that reaches the host processor. Therefore, we seek to provide intermediate levels of data transfer that enable some distributed inter-crossbar communication with parallelism across pairs of crossbars.

We propose the recursive H-tree structure seen in Figure 9 (for an example of 16 crossbar arrays) that enables distributed inter-crossbar communication since each group of the hierarchy is capable of either intra-group communication (in parallel to the other groups at the same level) or inter-group communication (to other groups at the same level or higher). We have chosen an H-tree structure as opposed to other communication frameworks due to the simplicity of its implementation and its generalizability to a wide range of applications since it enables inter-crossbar communication across various levels of parallelism and inter-crossbar distance. The numbering of the crossbars in the H-tree (which corresponds to the numbering for the crossbar mask operation) is constructed recursively where each group includes all of the crossbars that share a certain prefix (e.g., group  $10xx$  includes 1000, 1001, 1010 and 1011). Overall, the example structure in Figure 9 supports the following types of data movement:

- Data movement within each crossbar (using Section III-E) and in parallel across up to all crossbars.
- Data movement within each group of 4 crossbars, in parallel across up to all other groups. For example, XB 0001 transfers data to XB 0010 at the same time that XB 0101 transfers data to XB 0110 and so forth. We may formalize this example as crossbars  $xx01$  transferring data to crossbars  $xx10$  for all  $xx$ .

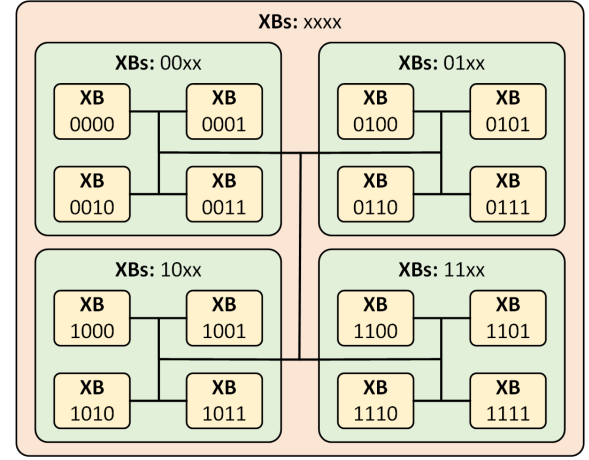


Fig. 9. An example hierarchical H-tree consisting of 16 crossbar arrays (XBs) numbered from 0000 to 1111. Each group of crossbars is characterized by crossbar indices with a shared prefix and differing suffixes (e.g., group  $10xx$  corresponds to crossbars 1000, 1001, 1010 and 1011).

- Data movement within the group of 16 crossbars, in parallel across other groups of 16 crossbars.

We formalize the distributed communication pattern for a set of crossbar pairs as follows. Consider the set of crossbars that constitute the input crossbar of each pair,  $XBs = \{XB_{start}, XB_{start} + XB_{step}, \dots, XB_{stop}\}$  where  $XB_{step}$  is a power of 4 and  $XB_{stop} - XB_{start}$  is a multiple of  $XB_{step}$ . Let  $XB_{dist}$  be the distance between the input and output crossbar of each pair – we restrict this distance to be uniform across all pairs. Therefore, we find that each crossbar  $XB$  in  $XBs$  will transfer data to crossbar  $XB + XB_{dist}$ . For example, the transfer from the previous paragraph (crossbars  $xx01$  transferring data to crossbars  $xx10$  for all  $xx$ ) will be formatted as  $XB_{start} = (0001)_2$ ,  $XB_{step} = (0100)_2$ ,  $XB_{end} = (1101)_2$  and  $XB_{dist} = (0010)_2 - (0001)_2$ .

We utilize the above formalization to represent move operations in the proposed microarchitecture. To perform a distributed inter-crossbar move operation, the crossbar mask is first set to match  $XB_{start}, XB_{step}, XB_{end}$  via a mask operation, and then a move operation is issued with  $XB_{dist}$ , the source/destination rows, and the intra-partition indices.<sup>2</sup> The input crossbars (identified as crossbars activated by the mask) essentially perform an  $N$ -bit read operation that outputs the result on the H-tree bus, the output crossbars perform a write operation, and interconnect switches control the connection between the groups according to  $XB_{step}$ .

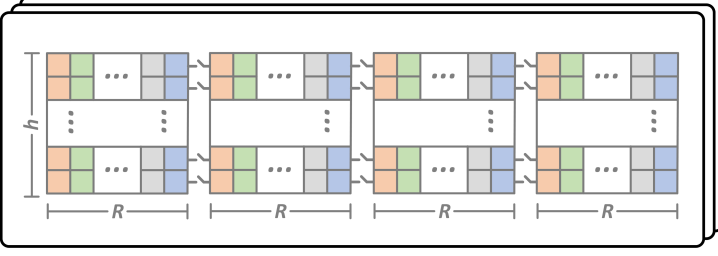
## IV. INSTRUCTION-SET ARCHITECTURE

We propose a general-purpose PIM instruction set architecture (ISA) that abstracts the implementation details of memristive digital PIM. This ISA represents the interface between the proposed PIM library and the host driver, and it enables the library to generalize to routines such as logarithmic reduction while supporting all PIM architectures with an appropriate

<sup>2</sup>We store  $XB_{dest} = XB_{start} + XB_{dist} \geq 0$  to avoid negative  $XB_{dist}$ .



(a) Crossbar Arrays



(b) Warps

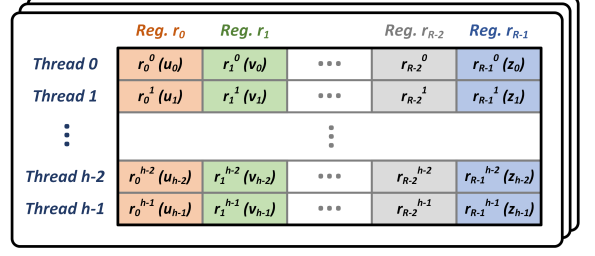


Fig. 10. The proposed ISA expresses (a) crossbar arrays as (b) warps of threads that may operate concurrently, where each register index may represent an  $h$ -dimensional vector (e.g., register  $r_{R-1}$  represents the vector  $\vec{z}$ ).

(a) Type I: R-type Instructions

	Reg. $r_0$	Reg. $r_1$	...	Reg. $r_{R-2}$	Reg. $r_{R-1}$
Thread 0	$u_0$	$v_0$	...	$u_0 + v_0$	$z_0$
Thread 1	$u_1$	$v_1$	...	$u_1 + v_1$	$z_1$
...	...	...	...	...	...
Thread $h-2$	$u_{h-2}$	$v_{h-2}$	...	$u_{h-2} + v_{h-2}$	$z_{h-2}$
Thread $h-1$	$u_{h-1}$	$v_{h-1}$	...	$u_{h-1} + v_{h-1}$	$z_{h-1}$

$r_{R-2} = r_0 + r_1$

(b) Type II: Move Instructions

	Reg. $r_0$	Reg. $r_1$	...	Reg. $r_{R-2}$	Reg. $r_{R-1}$
Thread 0	$u_0$	$v_0$	...		$z_0$
Thread 1	$u_1$	$v_1$	...		$z_1$
...	...	...	...	...	...
Thread $h-2$	$u_{h-2}$	$v_1$	...		$z_{h-2}$
Thread $h-1$	$u_{h-1}$	$v_{h-1}$	...		$z_{h-1}$

$r_1^{h-2} = r_1^1$

Fig. 11. The proposed ISA enables (a) R-type instructions (thread-parallel register operations) and (b) move instructions (inter-thread data transfer) either within warps or according to the pattern specified in Section III-F. Note:  $r_i$  refers to the register at index  $i$ , and  $u_i, v_i, z_i$  refer to values.

host driver. The proposed model is based on an abstraction of crossbar arrays as *warps of threads*, where each thread is a single row that contains  $R$   $N$ -bit registers,<sup>3</sup> as shown in Figure 10. Warps are capable of performing R-type (register) macro-instructions (referred to as “instructions” for simplicity) in parallel across (up to) all threads, or inter-thread move instructions. Furthermore, inter-warp communication is also possible through *move* operations between warps.

The proposed ISA differs from compute-oriented architectures (e.g., CUDA) in that the computation is performed *within* the memory – *the registers of the threads are also the memory itself*. For example, consider the task of performing arithmetic on vectors stored in the memory. CUDA would *allocate* threads (in dedicated CUDA cores) that copy the data from the memory to registers, perform the arithmetic on the registers, and then write the results back to the memory. Conversely, the proposed ISA would *activate* the existing threads representing the memory where the vectors are stored, and those threads would operate directly on the vectors through their registers. While this may dwarf data transfer, it also imposes significant limitations on the vector alignment in the memory.

Figure 11 illustrates the two instruction types of the proposed ISA: register (R-type) and move. Register-type instructions represent arithmetic operations on the registers in parallel across (up to) all threads simultaneously, where the activated threads follow the same flexible range-based pattern defined in Section III. We support the AritPIM [29] suite of arithmetic

functions (addition, subtraction, multiplication, and division) on both fixed-point and floating-point numbers. We further extend the ISA to include comparison operations, bitwise operations, and miscellaneous routines (e.g., absolute value, multiplexing); the full list of supported operations is available in Table II. Further, the ISA also supports warp-parallel thread-serial move instructions that enable communication between different threads in the same warp and communication between threads in different warps according to the inter-array move format from Section III-F. These instructions move a register value from one thread to a different thread when the registers are aligned, according to warp pairs that satisfy the requirements of Section III-F. Lastly, the ISA supports standard read and write instructions to the memory. Read instructions are targeted at a single register in a single thread of a single warp, while write instructions also target a single register yet may be repeated across several threads or warps following a range-based pattern (typically used for constants).

## V. PIM LIBRARY AND HOST DRIVER

We propose development and driver libraries that enable PIM programming with significant ease based on the proposed ISA and microarchitecture. The development library is a Python library that provides familiar tensor bindings to the proposed ISA and is responsible for higher-level algorithms such as dynamic memory management, whereas the driver efficiently translates the ISA macro-instructions from the development library into micro-operations.

<sup>3</sup>Where  $R$  is chosen at compile-time according to  $w \geq R \cdot N$ , and the ISA and microarchitecture share the same word size ( $N$ ).

TABLE II  
SUPPORTED R-TYPE OPERATIONS IN THE PROPOSED ISA.

Operation	Integer Support	Float Support
<b>Arithmetic</b>		
Addition	✓	✓
Subtraction	✓	✓
Multiplication	✓	✓
Division	✓	✓
Modulo	✓	
Negation	✓	✓
<b>Comparison</b>		
Less than (or equal to)	✓	✓
Greater than (or equal to)	✓	✓
Equal	✓	✓
<b>Bitwise</b>		
Bitwise Not	✓	✓
Bitwise And	✓	✓
Bitwise Or	✓	✓
Bitwise Xor	✓	✓
<b>Miscellaneous</b>		
Sign	✓	✓
Zero	✓	✓
Abs	✓	✓
Mux	✓	✓

### A. Development Library

The proposed development library enables the seamless integration of PIM into traditional tensor-based Python programs using libraries such as NumPy [20] and PyTorch [36]. This both provides a familiar Python programming interface for PIM, and also enables PIM to be easily integrated within larger applications (e.g., hybrid CPU-PIM or CPU-GPU-PIM development). The library comprises the following components:

- *Python Bindings*: The library includes Python tensor-based bindings that enable the development of *parallel* digital PIM applications using the flexibility and simplicity of Python. The example program from Figure 2 demonstrates the support for initialization of PIM vectors (e.g., `pim.zeros(1024)`), direct read/write access (e.g., `x[4] = 8.0`), the passing of PIM vectors as arguments (e.g., `myFunc`), and parallel arithmetic (e.g., `x * y`).
- *Dynamic Memory Management*: One of the largest challenges with digital PIM is the need for the memory to be properly aligned to enable parallelism; for example, the vectors  $x$  and  $y$  must be stored in the same warp for  $x * y$  to be computed. The development library addresses this challenge through the combination of (1) a fall-back routine that copies vectors if they are not in the same warp (to avoid throwing an error), and (2) the careful management of PIM vectors that attempts to allocate them consecutively. Specifically, PIM vectors are allocated at a specific register index across the rows of potentially several warps, and the `malloc` routine in the development library aims to map consecutive requests to the same warp ranges. We support the option of providing a *reference tensor* when allocating a tensor, whereby the memory allocation algorithm will first attempt to allocate memory aligned with the reference tensor.
- *Views and Data Movement*: We implement the slicing

operation (i.e., `[a:b:c]` in Python represents a slice of all numbers from inclusive  $a$  to exclusive  $b$  in steps of  $c$ ) using a concept of “tensor views” which represent the same underlying memory. For example, when performing `y = x[:,2]` (select all even elements) on tensor  $x$  then python object  $y$  is returned which represents the same underlying memory as  $x$  yet any operation on  $y$  will automatically invoke a row mask corresponding to all even rows (e.g., accessing `y[4]` leads to a memory read for the underlying memory of `x[8]`). We generalize the inter-warp data movement supported by the ISA using these views as inter-view data-transfer is automatically converted to the underlying move operations. For example, performing `x[:,2] + x[1:,2]` (sum of even and odd elements for a resulting tensor half the size) leads to the development library automatically identifying the move operations required to align the values before the addition (e.g., move operations will copy the contents of `x[1:,2]` to a new tensor aligned with `x[:,2]` and only then perform addition). This both enables data sharing with minimal effort, and also may serve as a powerful abstraction of inter-warp communication.

### B. Host Driver

The host driver is responsible for the translation of the abstract macro-instructions (e.g., register add) into the micro-operations that adhere to the proposed microarchitecture (e.g., logic NOR). While previous works implement this step in a dedicated on-chip controller [19], [39], [50], we propose an efficient host program that is not a bottleneck to PIM performance (see Section VI). The efficiency is due to the combination of efficient low-level implementations of the AritPIM [29] suite of algorithms,<sup>4</sup> and the compact representation for partition operations (Section III) that maintains moderate data transfer between the host driver and the on-chip controller. While a dedicated on-chip hardware controller would further reduce data transfer to only the encoding of the macro-instruction rather than all of the micro-operations, this would require custom controller circuitry and serve as barrier to future changes to the controller (in contrast to a software driver that may be readily updated). The development library and ISA proposed in PyPIM may also be directly applied to different digital PIM architectures by replacing the host driver, *thereby enabling the same high-level PIM application to be applicable to a wide variety of digital PIM architectures*. Moreover, manufacturers of digital PIM architectures may simply adapt the PyPIM host driver according to their technologies.

### C. End-to-End Integration

We analyze the end-to-end integration of PyPIM through the example program in Figure 12. The program is developed in Python and utilizes the `pim.Tensor` class as a standalone

<sup>4</sup>The algorithms were modified slightly to accommodate additional cases (e.g., signed integer division rather than only unsigned) according to the recommended extensions discussed in the AritPIM [29] library. Further, the integer multiplication algorithm was truncated to output 32 bits instead of 64.

```

1 import pypim as pim
2 def myFunc(a: pim.Tensor, b: pim.Tensor):
3     # Parallel multiplication and addition
4     return a * b + a
5
6 # Tensor initialization
7 x = pim.zeros(2 ** 20, dtype=pim.float32)
8 y = pim.zeros(2 ** 20, dtype=pim.float32)
9 x[4], y[4] = 8.0, 0.5
10 x[5], y[5] = 20.0, 1.0
11 x[8], y[8] = 10.0, 1.0
12
13 # Custom function call
14 z = myFunc(x, y)
15 # Logarithmic-time reduction of even indices
16 print(z[:, :2].sum()) # 32.0 = 8 * 1.5 + 10 * 2

```

Fig. 12. An example application for the end-to-end integration of PyPIM.

replacement for the familiar `numpy.array` class, with the additional functionality of element-wise operations that are automatically translated into digital PIM micro-operations.

- `pim.zeros(2 ** 20, dtype=pim.float32)`: The development library allocates 1M-element floating-point vectors  $x$  and  $y$  and initializes them to zeros. Specifically, the `malloc` routine is executed and also a write macro-instruction is transmitted to the host driver (translated into the mask and write micro-operations).
- `x[4] = 8.0; y[4] = 1.0`: These operations are translated by the programming interface into write macro-instructions that are then translated by the host driver into mask and write micro-operations.
- `myFunc(x, y)`: The function call passes the tensors by reference, identical to the behavior of `numpy.array`. Within the function, Python first performs  $a * b$  by having the development library first allocate memory for the output tensor and then call the host driver to perform the multiplication instruction on the given memory addresses. The address of that intermediate is then used to perform addition with  $a$ , leading to the desired result.
- `z[:, :2]`: The function call receives a given tensor object and returns a view of the tensor that represents all even values. While the returned tensor view possesses the same data pointer for the memory, the elements in the tensor are accessed with the view's strides in order to replicate the behavior of a cloned tensor of even values (i.e., accessing the fourth element in the tensor will lead to the eighth element of the memory address being retrieved).
- `z[:, :2].sum()`: The development library receives the tensor *view* and then performs a logarithmic reduction operation on the vector to compute the overall sum. For example, in the case of `z[:, :2].sum()`, the first step is to compute the summation of sub-tensors `z[:, :4].sum()` and `z[2:, :4].sum()`. This summation between non-aligned vectors is first automatically mapped to intra- and inter-warp move operations by the development library and is then followed by traditional summation on aligned tensors. This process repeats recursively until the last element is retrieved as the summation of all elements.

TABLE III  
THE EVALUATION PARAMETERS FOR PYPIM.

System	Parameters
Host	CPU: 2x AMD EPYC 7513 (32 core) GPU: NVIDIA A6000 Memory Size: 16x 64GB Memory BW (per chip): 25.6 GB/s
Simulated PIM [29]	Memory Size: 8GB Crossbars: 1024 × 1024 (32 partitions) Word Size (N): 32 Clock Frequency: 300 MHz

## VI. EVALUATION

We evaluate PyPIM to demonstrate the correctness and performance of the proposed mechanisms as compared to theoretical PIM results. To that end, we first develop a bit-accurate GPU-accelerated digital PIM simulator that is a drop-in replacement for a digital PIM chip as it performs micro-operations on an internal simulated memory in the same manner as performed by a memristive PIM chip. This enables correctness verification by (1) loading the memory with sample data, (2) performing the PyPIM algorithms with the micro-operations generated by the host driver being forwarded to the simulator, and (3) reading the results and comparing with expected values (e.g., computed using standard CPU libraries). Further, the simulator keeps track of basic profiling metrics (e.g., the number of micro-operations performed from each micro-operation type) that are then used to compare the performance of step (2) with the expected theoretical performance. Table III lists the parameters used in the evaluation, including both the host system on which the libraries are executed and the parameters for the digital PIM architecture.

### A. Correctness

We evaluate the correctness of the proposed end-to-end integration from the high-level Python code to the generated sequence of micro-operations through a bit-level PIM simulator. The simulator is a drop-in replacement for a digital PIM chip, thereby verifying that the proposed libraries lead to the intended results when paired with a digital PIM architecture. We follow the previously-established standard of cycle-accurate simulation [17], [28]–[30], [35], which requires that (1) the only interaction between the simulator and the library is through the interface for micro-operations, (2) the simulator models the operations cycle-by-cycle and executes them the same as a digital PIM chip, and (3) the results are compared to the intended output as generated by a trusted CPU-only program. Since we are simulating an entire PIM memory (in contrast to previous works that typically only simulated single rows or crossbars), we also accelerate the simulation time by using a CUDA-enabled GPU and exploiting two optimizations:

- *Memory*: The simulator stores the state of the digital PIM chip in the GPU memory – the logical states of all rows of all crossbars. Whereas previous works stored each row as a vector of Booleans (effectively stored as bytes) [28]–

[30], [35], we store rows in a condensed 32-bit format that is defined according to the strided data format.

- *Logic*: We perform the logic operations efficiently by exploiting CUDA bitwise arithmetic operations to perform semi-parallel and parallel operations (rather than iterating over the partitions explicitly). Furthermore, the micro-operations are performed in batches to improve the memory locality of each crossbar in the simulator.

We evaluate PyPIM on the following set of benchmarks that test the different tensor operations supported by the development library (available in the code repository [1]) and their extension to inter-crossbar algorithms:

- *Fundamental Arithmetic*: The four fundamental arithmetic operations (addition, subtraction, multiplication, and division) performed on randomly-generated integer and floating-point tensors.
- *Comparison Operations*: The supported comparison operations (less than, less than or equal to, greater than, greater than or equal to, equal to, and not equal to) with randomly-generated integer and floating-point tensors.
- *CORDIC Sine/Cosine*: We utilize the Python interface of the PyPIM library to construct sine/cosine approximations using the CORDIC algorithm [51]. We evaluate these algorithms on randomly-generated floating-point tensors in the range  $[-\pi/2, \pi/2]$ .
- *Reduction*: We implement logarithmic reduction [41] using the Python interface of the PyPIM library, effortlessly performing the data movements (both intra- and inter-crossbar) through the “Tensor views” functionality (see Section V-A). We evaluate reduction with both summation and multiplication across randomly-generated integer and floating-point tensors.
- *Sorting*: Similar to the reduction benchmark, we utilize the Python interface of PyPIM alongside the supported “Tensor views” to implement intra- and inter-crossbar sorting in only 18 lines of Python code. We utilize a bitonic sorting network [5] that expresses sorting as a sequence of parallel compare-and-swap operations (receive two numbers  $a, b$  and output  $\min(a, b), \max(a, b)$ ) followed by data movement between elements.

We execute these benchmarks with the proposed simulator and compare the results with those generated by the CPU-based NumPy [20] library (which adheres to the IEEE 754 floating-point standards). Further, we include the implementation of these algorithms in the PyPIM library package so that they may be used directly in other works (e.g., `x.sort()`).

### B. Throughput

We verify that the proposed framework attains the potential throughput provided by digital PIM for in-memory logic. We measure the performance of the framework on the unit tests discussed in the previous subsection, whereby we evaluate the following two characteristics:

- *PIM Cycle Count*: We compare the number of PIM cycles (number of micro-operations) performed for the unit test

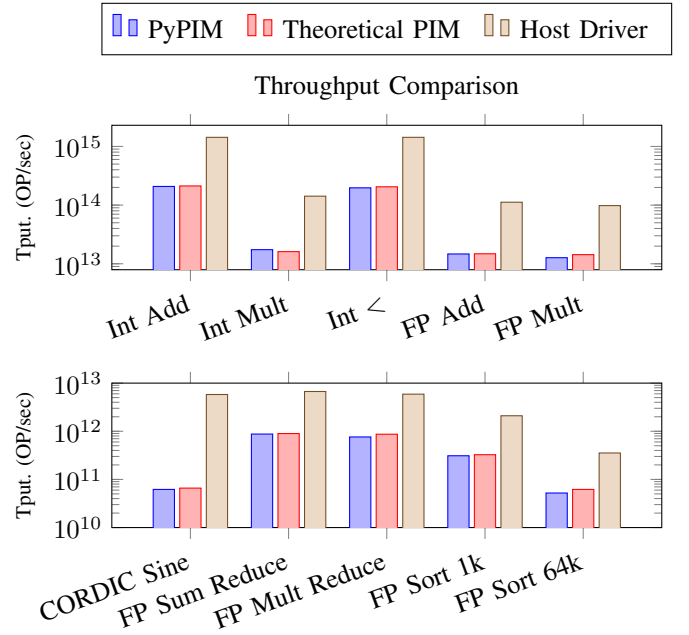


Fig. 13. Overview of the results across the different benchmarks for the PyPIM library and the theoretical PIM performance (according to Table III), as well as the theoretical maximal throughput supported by the host driver (demonstrating that the host driver is not a bottleneck).

to the theoretical lower-bound required based on previous works (e.g., AritPIM).

- *Host Driver Runtime*: To verify that the host driver does not bottleneck the PIM performance, we evaluated the maximal supported throughput of the host driver and compared it to the expected PIM throughput. This demonstrates the claim that a hardware controller is not necessary due to the efficiency of the host driver.

Figure 13 presents these results across the benchmarks from the previous subsection,<sup>5</sup> displaying for each (1) the PIM performance derived from the latency metrics collected by the simulator, (2) the PIM performance derived from the theoretical latency, and (3) the maximal performance that can be supported by the host driver. We find that PyPIM is on average (worst case) 5% (16%) away from theoretical PIM, and that the host driver is on average (worst case)  $9.5\times$  ( $6.8\times$ ) faster than PyPIM (demonstrating that it is not a bottleneck).

## VII. ADDITIONAL CONSIDERATIONS

We present in this section several miscellaneous considerations and limitations of PyPIM.

- 1) **Virtual Memory**: While PyPIM does not explicitly support memory virtualization, future work may add this capability by either: (1) providing software virtualization through the PyPIM libraries, or (2) utilizing the same hardware mechanisms present in traditional memory

<sup>5</sup>Due to limited space, the Figure only includes a few representation benchmarks (e.g., only “less than” from the “comparison” benchmarks) with the full results available in the code repository [1].



virtualization after the controller – where each page corresponds to a single warp (e.g., 128KB pages).

- 2) **Reliability and Endurance:** There is an ongoing effort to address the reliability and endurance of memristors for digital PIM. PyPIM does not exacerbate the problem with either reliability or endurance, and such reliability techniques can also be integrated into the host driver. Regardless, the same concepts proposed in PyPIM may be applied to other architectures with improved reliability or endurance such as DRAM PIM [14], [19], [42].
- 3) **Parameter Exposure:** While the proposed ISA abstracts away the low-level details of the PIM architectures, the crossbar array dimensions dictate the number of registers per thread and the number of threads per warp. Further, PyPIM requires that the internal mechanisms and parameters of the digital PIM architecture (e.g., the supported logic gates) be exposed to the host driver. If the manufacturer does not intend to reveal this information, then the manufacturer may provide an obfuscated [4] driver that encodes this information.
- 4) **Technology Simulations:** PyPIM covers the integration of PIM from high-level Python instructions to the low-level micro-operations that are issued to the memory arrays. Future work may further continue through the integration with lower-level circuit simulations of the underlying memory technology. Such simulations may be integrated with PyPIM by (1) passing the micro-operations generated by the host driver to the simulation rather than the current GPU-based logical PIM simulator, and (2) integrating the circuitry detailed in Section III as part of the simulations.

## VIII. CONCLUSION

As digital PIM architectures continue to emerge, there has been significant work on the algorithmic potential of the underlying parallel computing model – from high-throughput arithmetic to large-scale applications. This paper provides the first end-to-end architectural integration of digital PIM from the low-level micro-operation format to a familiar tensor-based Python programming interface. This is accomplished by first proposing a microarchitecture for partition-enabled digital memristive PIM and an abstract instruction set architecture (ISA), and then developing libraries that enable parallel vectored PIM operations in Python programs with significant ease. We further propose a GPU-accelerated simulator that interfaces with the driver to verify correctness and enable the efficient testing of PIM applications. Overall, PyPIM increases the accessibility of PIM to the wider community through a familiar programming interface with a powerful abstraction.

## IX. ACKNOWLEDGMENTS

This work was supported by the European Research Council through the European Union’s Horizon 2020 Research and Innovation Programme under Grants 757259 and 101069336. This research received partial support from the Cisco University Research Program Fund.

## A. Abstract

The artifact repository includes the framework proposed in the paper and the additional tests and benchmarks that consist of the evaluation of the framework. The framework enables high-level programming of PIM applications with significant ease as it follows the same abstraction as existing tensor-based Python libraries (e.g., NumPy, PyTorch, TensorFlow) to provide the user with a familiar interface. We utilize a CUDA-accelerated cycle-accurate simulator of the PIM framework in lieu of a physical PIM chip (requiring an NVIDIA GPU in order to test the framework) to enable the library to be tested and debugged today. The goal of this artifact repository is both to facilitate the installation and usage of the framework in order to support its integration in future works, and to present the tests and benchmarks required to replicate the key results from the work. Expected output for all the tests and benchmarks is also provided in the repository.

## B. Artifact check-list (meta-information)

- **Compilation:** CUDA Toolkit and C++ compiler.
- **Run-time environment:** CUDA and Python.
- **Hardware:** NVIDIA GPU with at least 9GB DRAM.
- **Output:** The provided tests verify correctness and performance, with expected output also provided.
- **How much disk space required (approximately)?:** < 30GB.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** Approximately 4 hours.
- **Publicly available?:** Yes.
- **Archived:** <https://doi.org/10.5281/zenodo.13733284>.

## C. Description

1) *How to access:* The source code can be cloned from the GitHub repository [1].

2) *Hardware dependencies:* CUDA-capable NVIDIA GPU with at least 8GB DRAM. Tested on NVIDIA A6000.

3) *Software dependencies:*

- CUDA Toolkit (at least 12.0)
- Compiler for C++ 17 (compatible with CUDA version).
- Python and pip installation (tested with 3.10)

Alternatively, we recommend using the PyTorch docker container [2] as it contains all of the required dependencies.

4) *Data sets:* None are required as performance and correctness are measured on randomly-generated data created as part of the tests.

## D. Installation

Build and install the library by running the following from the root directory of the repository:

```
1 $ pip install -e .
```

Test if the installation succeeded by attempting to import the Python library and verifying that there are no errors:

```
1 $ python
2 >>> import pypim as pim
3 >>>
```

### E. Experiment workflow

The following three commands replicate the correctness and performance measurements of PyPIM (see Section VI and Figure 13 for further details), with their expected output available in the *results* folder of the repository:

- *Unit tests*: `python tests/unit.py`
- *Reduction tests*: `python tests/reduction.py`
- *Sort tests*: `python tests/sort.py`

To derive the maximal supported PyPIM throughput from the latency reported by the simulator we use the following equation:

$$\begin{aligned} \text{Throughput}[\text{ops/sec}] &= \\ \text{Throughput}[\text{ops/cycle}] \cdot \text{Frequency}[\text{cycles/sec}] &= \\ \frac{\text{Parallelism}[\text{ops}]}{\text{Latency}[\text{cycles}]} \cdot \text{Frequency}[\text{cycles/sec}] &= \end{aligned} \quad (1)$$

where  $\text{Parallelism}[\text{ops}]$  is the number of rows of the crossbar memory (e.g., 64M according to Table III).

The performance of the host driver itself is evaluated by simulating the conditions of an ideal digital PIM chip and focusing on what is the maximal throughput of PIM operations that the PIM chip may operate at while having the host driver generate the micro-operations (i.e., what is the maximal PIM throughput supported by the host driver if the PIM chip was not the bottleneck). To that end, we model this situation by rerouting the micro-operations to a memory buffer instead of the GPU-based simulator, thereby measuring the performance of the CPU operations of the host driver involved in the micro-operation generation. Specifically, we modify the `csrc/driver/driver.cpp` file as follows:

- 1) Modify the imports as follows and have the perform function of the simulator be overridden with a write to a memory buffer (`__LINE__` is used as a sequence of consecutive numbers so that each micro-operation of an instruction is written to a different address).

```
1 // #include <pybind11/pybind11.h>
2 #include <iostream>
3 #include <chrono>
4 #include "driver.h"
5 #include "simulator.cuh"
6
7 pim::otype OPS[100000];
8 #define perform(x) OPS[__LINE__] = x;
```

- 2) Comment out the code block starting at line `PYBIND11_MODULE(driver, m)` and add the benchmark test afterwards. For example, for the benchmark of integer addition, we add:

```
1 int main()
2 {
3     pim::otype res = 0;
4     std::chrono::steady_clock::time_point
5         start = std::chrono::steady_clock::
6             now();
7     for (long i = 0; i < 100000001; i++)
8     {
9         pim::add<int>(rand() % 32, rand() %
10             32, rand() % 32, pim::
11             driverCrossbarMask, pim::
12             driverCrossbarMask);
```

```
8         res += OPS[rand() % 100000];
9     }
10     std::chrono::steady_clock::time_point
11         end = std::chrono::steady_clock::
12             now();
13     std::cout << "Time difference = " << std
14         ::chrono::duration_cast<std::chrono
15         ::microseconds>(end - start).count
16         () << "[us]" << std::endl;
```

The other supported macro-instructions can be performed by replacing e.g. `pim::add` with `pim::multiply` or `pim::divide`, and replacing `<int>` with `<float>`.

- 3) Compile from within the `csrc/driver` directory with

```
1 g++ driver.cpp -I ../ -I ../simulator/ -O3 -
  o driver_benchmark
```

and run `./driver_benchmark`.

### F. Evaluation and expected results

The expected results of the above commands are available in the *results* directory of the repository (the exact output of the tests, reflecting the performance measurements from the GPU-based simulator). The provided tests only succeed if the correctness comparison of the framework with the expected ground-truth result (based on NumPy) is successful; for example, see the *test\_arit* test (from the *tests/unit.py* file) which verifies the correctness of all arithmetic operations on integer and floating-point numbers:

```
1 @parameterized.expand([
2     ('__add__', np.add, np.dtype(np.int32)),
3     ('__sub__', np.subtract, np.dtype(np.int32)),
4     ('__mul__', np.multiply, np.dtype(np.int32)),
5     ('__truediv__', np.true_divide, np.dtype(np.
6         int32)),
7     ('__add__', np.add, np.dtype(np.float32)),
8     ('__sub__', np.subtract, np.dtype(np.float32)),
9     ('__mul__', np.multiply, np.dtype(np.float32)),
10    ('__truediv__', np.true_divide, np.dtype(np.
11        float32))
12 ])
13 def test_arit(self, function, gt_func, dtype):
14     print(f'Testing arithmetic with {function} on
15         type {dtype}')
16     nelelem = 2 ** 16
17     ninputs = 2
18
19     # Allocate and initialize the tensors
20     refs = [utils.rand(nelelem, dtype) for _ in
21         range(ninputs)]
22     tensors = [pim.from_numpy(x) for x in refs]
23
24     # Perform the arithmetic function
25     with pim.Profiler():
26         result = getattr(tensors[0], function)(
27             tensors[1])
28     result = pim.to_numpy(result)
29
30     # Compare to ground-truth
31     ground_truth = gt_func(refs[0], refs[1]).
32         astype(dtype)
33     np.testing.assert_array_equal(ground_truth,
34         result)
```

## G. Experiment customization

The experiments may be customized through modifications to the testing parameters such as `nlem` in the above scripts. We further encourage the overall experimentation with the library in an interactive format as with existing tensor-based Python libraries, or using custom tests by following the same file structure as the provided tests. For example, see the following interactive use of the framework through the standard Python interpreter which tests tensor allocation, read/write operations, tensor views, summation reduction and sorting:

```
1 >>> import pypim as pim
2 >>> x = pim.zeros(8, dtype=pim.float32)
3 >>> x
4 Tensor(shape=(8,), dtype=float32):
5 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
6 >>>
7 >>> x[2] = 2.5
8 >>> x[3] = 1.25
9 >>> x[4] = 2.25
10 >>> x
11 Tensor(shape=(8,), dtype=float32):
12 [0.0, 0.0, 2.5, 1.25, 2.25, 0.0, 0.0, 0.0]
13 >>>
14 >>> x[:2]
15 TensorView(shape=(4,), dtype=float32, slicing=
16 slice(0, 7, 2)):
17 [0.0, 2.5, 2.25, 0.0]
18 >>> x[:2].sum()
19 4.75
20 >>> x[:2].sort()
21 TensorView(shape=(4,), dtype=float32, slicing=
22 slice(0, 7, 2)):
23 [0.0, 0.0, 2.25, 2.5]
```

## H. Notes

The derivation of the throughput results of Figure 13 follows from the latency and gate count metrics produced by the repository according to the constants from Table III.

## I. Methodology

Submission, reviewing and badging methodology:

- <https://www.acm.org/publications/policies/artifact-review-and-badging-current>
- <https://cTuning.org/ae>

## REFERENCES

- [1] “PyPIM GitHub Repository,” <https://github.com/oleitersdorf/PyPIM>.
- [2] “PyTorch Docker Container,” <https://hub.docker.com/r/pytorch/pytorch>.
- [3] M. R. Alam, M. H. Najafi, and N. TaheriNejad, “Sorting in memristive memory,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 18, no. 4, 2022.
- [4] S. Banescu and A. Pretschner, “A tutorial on software obfuscation,” ser. *Advances in Computers*, A. M. Memon, Ed. Elsevier, 2018, vol. 108, pp. 283–353.
- [5] K. E. Batcher, “Sorting networks and their applications,” in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS ’68 (Spring). New York, NY, USA: Association for Computing Machinery, 1968, p. 307–314.
- [6] R. Ben Hur and S. Kvatinisky, “Memristive memory processing unit (MPU) controller for in-memory processing,” in *IEEE International Conference on the Science of Electrical Engineering (ICSEE)*, 2016.
- [7] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinisky, “SIMPLER MAGIC: Synthesis and mapping of in-memory logic executed in a single row to improve throughput,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 10, pp. 2434–2447, 2020.
- [8] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, “‘Memristive’ switches enable ‘stateful’ logic operations via material implication,” *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.
- [9] Z. Chowdhury, J. D. Harms, S. K. Khatamifard, M. Zabihi, Y. Lv, A. P. Lyle, S. S. Sapatnekar, U. R. Karpuzcu, and J.-P. Wang, “Efficient in-memory processing using spintronics,” *IEEE Computer Architecture Letters (CAL)*, vol. 17, no. 1, pp. 42–46, 2018.
- [10] L. Chua, “Memristor – the missing circuit element,” *IEEE Transactions on Circuit Theory (TCT)*, vol. 18, no. 5, pp. 507–519, 1971.
- [11] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, “Neural cache: Bit-serial in-cache acceleration of deep neural networks,” in *ACM/IEEE International Symposium on Computer Architecture*, 2018, pp. 383–396.
- [12] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie, “Computational RAM: implementing processors in memory,” *IEEE Design & Test of Computers*, vol. 16, no. 1, pp. 32–41, 1999.
- [13] D. Fujiki, S. Mahlke, and R. Das, “Duality cache for data parallel acceleration,” in *ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 1–14.
- [14] F. Gao, G. Tziantzioulis, and D. Wentzlaff, “ComputeDRAM: In-memory compute using off-the-shelf DRAMs,” in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.
- [15] S. Gupta, M. Imani, B. Khaleghi, V. Kumar, and T. Rosing, “RAPID: A ReRAM processing in-memory architecture for DNA sequence alignment,” in *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2019.
- [16] S. Gupta, M. Imani, and T. Rosing, “FELIX: Fast and energy-efficient logic in memory,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.
- [17] A. Haj-Ali, R. Ben-Hur, N. Wald, and S. Kvatinisky, “Efficient algorithms for in-memory fixed point multiplication using MAGIC,” in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018.
- [18] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinisky, “IMAGING: In-memory algorithms for image processing,” *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS-I)*, vol. 65, no. 12, pp. 4258–4271, 2018.
- [19] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. a. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, “SIM-DRAM: A framework for bit-serial SIMD processing using DRAM,” in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.
- [20] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.
- [21] B. Hoffer, V. Rana, S. Menzel, R. Waser, and S. Kvatinisky, “Experimental demonstration of memristor-aided logic (MAGIC) using valence change memory (VCM),” *IEEE Transactions on Electron Devices (TED)*, vol. 67, no. 8, pp. 3115–3122, 2020.
- [22] B. Hoffer, N. Wainstein, C. M. Neumann, E. Pop, E. Yalon, and S. Kvatinisky, “Stateful logic using phase change memory,” *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 8, no. 2, pp. 77–83, 2022.
- [23] M. Horowitz, “Computing’s energy problem (and what we can do about it),” in *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014.
- [24] M. Imani, S. Gupta, Y. Kim, and T. Rosing, “FloatPIM: In-memory acceleration of deep neural network training with high precision,” in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2019.
- [25] H. Jin, C. Liu, H. Liu, R. Luo, J. Xu, F. Mao, and X. Liao, “ReHy: A ReRAM-based digital/analog hybrid PIM architecture for accelerating CNN training,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 11, pp. 2872–2884, 2022.
- [26] M. Khalifa, R. Ben-Hur, R. Ronen, O. Leitersdorf, L. Yavits, and S. Kvatinisky, “FiltPIM: In-memory filter for DNA sequencing,” in *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2021.

- [27] S. Kvatsinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC – memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs (TCAS-II)*, vol. 61, no. 11, pp. 895–899, 2014.
- [28] O. Leitersdorf, Y. Boneh, G. Gazit, R. Ronen, and S. Kvatsinsky, "FourierPIM: High-throughput in-memory fast fourier transform and polynomial multiplication," *Memories – Materials, Devices, Circuits and Systems*, vol. 4, p. 100034, 2023.
- [29] O. Leitersdorf, D. Leitersdorf, J. Gal, M. Dahan, R. Ronen, and S. Kvatsinsky, "AritPIM: High-throughput in-memory arithmetic," *IEEE Transactions on Emerging Topics in Computing (TETC)*, pp. 1–16, 2023.
- [30] O. Leitersdorf, R. Ronen, and S. Kvatsinsky, "MatPIM: Accelerating matrix operations with memristive stateful logic," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2022.
- [31] —, "MultPIM: Fast stateful multiplication for processing-in-memory," *IEEE Transactions on Circuits and Systems II: Express Briefs (TCAS-II)*, vol. 69, no. 3, pp. 1647–1651, 2022.
- [32] R. Li, S. Song, Q. Wu, and L. K. John, "Accelerating force-directed graph layout with processing-in-memory architecture," in *IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2020.
- [33] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-based reconfigurable in-situ accelerator," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.
- [34] Z. Lu, M. T. Arafat, and G. Qu, "RIME: A scalable and energy-efficient processing-in-memory architecture for floating-point operations," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021.
- [35] B. Oved, O. Leitersdorf, R. Ronen, and S. Kvatsinsky, "HashPIM: High-throughput SHA-3 via memristive digital processing-in-memory," in *IEEE International Conference on Modern Circuits and Systems Technologies (MOCAS)*, 2022.
- [36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, 2019, pp. 8024–8035.
- [37] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.
- [38] A. Pedram, S. Richardson, M. Horowitz, S. Galal, and S. Kvatsinsky, "Dark memory and accelerator-rich system optimization in the dark silicon era," *IEEE Design & Test*, vol. 34, no. 2, pp. 39–50, 2017.
- [39] B. Perach, R. Ronen, B. Kimelfeld, and S. Kvatsinsky, "Understanding bulk-bitwise processing in-memory through database analytics," *IEEE Transactions on Emerging Topics in Computing*, vol. 12, no. 1, pp. 7–22, 2024.
- [40] J. Reuben, R. Ben-Hur, N. Wald, N. Talati, A. H. Ali, P.-E. Gaillardon, and S. Kvatsinsky, "Memristive logic: A framework for evaluation and comparison," in *International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017.
- [41] R. Ronen, A. Eliahu, O. Leitersdorf, N. Peled, K. Korgaonkar, A. Chatopadhyay, B. Perach, and S. Kvatsinsky, "The bitlet model: A parameterized analytical model to compare PIM and CPU systems," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 18, no. 2, 2022.
- [42] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.
- [43] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, 2008.
- [44] Z. Sun, E. Ambrosi, A. Bricalli, and D. Ielmini, "Logic computing with stateful neural networks of resistive switches," *Advanced Materials*, 2018.
- [45] Z. Sun, S. Kvatsinsky, X. Si, A. Mehonic, Y. Cai, and R. Huang, "A full spectrum of computing-in-memory technologies," *Nature Electronics*, vol. 6, no. 11, pp. 823–835, 2023.
- [46] N. Talati, "Design of a memory controller to support PIM operations using RRAM," Master's thesis, Viterbi Faculty of Electrical Engineering, Technion – Israel Institute of Technology, Haifa, Israel, 2018.
- [47] N. Talati, R. Ben-Hur, N. Wald, A. Haj-Ali, J. Reuben, and S. Kvatsinsky, "mMPU – a real processing-in-memory architecture to combat the von Neumann bottleneck," *Applications of Emerging Memory Technology: Beyond Storage*, pp. 191–213, 2020.
- [48] N. Talati, S. Gupta, P. Mane, and S. Kvatsinsky, "Logic design within memristive memories using memristor-aided logic (MAGIC)," *IEEE Transactions on Nanotechnology (TNANO)*, vol. 15, no. 4, pp. 635–650, 2016.
- [49] N. Talati, H. Ha, B. Perach, R. Ronen, and S. Kvatsinsky, "CONCEPT: A column-oriented memory controller for efficient memory and PIM operations in RRAM," *IEEE Micro*, vol. 39, no. 1, pp. 33–43, 2019.
- [50] M. S. Q. Truong, E. Chen, D. Su, L. Shen, A. Glass, L. R. Carley, J. A. Bain, and S. Ghose, "RACER: Bit-pipelined processing using resistive memory," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.
- [51] J. E. Volder, "The cordic trigonometric computing technique," *IRE Transactions on electronic computers*, no. 3, pp. 330–334, 1959.
- [52] N. Wald and S. Kvatsinsky, "Understanding the influence of device, circuit and environmental variations on real processing in memristive memory using memristor aided logic," *Microelectronics Journal*, vol. 86, pp. 22–33, 2019.
- [53] Z. Wang, C. Liu, and T. Nowatzki, "Infinity stream: Enabling transparent and automated in-memory computing," *IEEE Computer Architecture Letters*, vol. 21, no. 2, pp. 85–88, 2022.
- [54] N. Xu, T. Park, K. J. Yoon, and C. S. Hwang, "In-memory stateful logic computing using memristors: Gate, calculation, and application," *Physica Status Solidi (RRL) – Rapid Research Letters*, vol. 15, no. 9, p. 2100208, 2021.