AritPIM: High-Throughput In-Memory Arithmetic

Orian Leitersdorf, *Student Member, IEEE,* Dean Leitersdorf, Jonathan Gal, Mor Dahan, Ronny Ronen, *Fellow, IEEE,* and Shahar Kvatinsky, *Senior Member, IEEE*

Abstract—Digital processing-in-memory (PIM) architectures are rapidly emerging to overcome the memory-wall bottleneck by integrating logic within memory elements. Such architectures provide vast computational power within the memory itself in the form of parallel bitwise logic operations. We develop novel *algorithmic* techniques for PIM that, combined with new perspectives on computer arithmetic, extend this bitwise parallelism to the four fundamental arithmetic operations (addition, subtraction, multiplication, and division), for both fixed-point and floating-point numbers, and using both bit-serial and bit-parallel approaches. We propose a state-of-the-art suite of arithmetic algorithms, demonstrating the first algorithm in the literature of digital PIM for a majority of cases – including cases previously considered impossible for digital PIM, such as floating-point addition. Through a case study on memristive PIM, we compare the proposed algorithms to an NVIDIA RTX 3070 GPU and demonstrate significant throughput and energy improvements.

Index Terms—Digital processing-in-memory (PIM), parallel computation, arithmetic, fixed-point arithmetic, floating-point arithmetic.

1 Introduction

Emerging processing-in-memory (PIM) systems attempt to overcome the memory-wall bottleneck by rethinking one of the core principles of computing systems: the separation of storage and logic units. This separation has been followed since the introduction of the von Neumann architecture in the 1940s, when computing systems were primarily utilized for *serial* program execution. Yet, the recent emergence of data-intensive applications requires *parallel high-throughput* execution, causing the separation to become a massive bottleneck known as the *memory wall* [1]. Therefore, PIM integrates logic within the memory itself to bypass the bandwidth-limited memory interface and enable massive in-memory computational parallelism [2].

PIM architectures supplement the traditional read/write memory interface with logic [2]. This enables the CPU to request that the memory perform *vectored* logic on data stored within the memory without transferring the data through the interface, thereby significantly reducing the load on CPU-memory communication. Early proposals for PIM [3] involved integrating logic circuits *near* the memory (e.g., in the same chip), yet this still requires a fundamental need for data-transfer between an area dedicated for computation and an area dedicated for storage [2]. Conversely, recent proposals [4], [5], [6], [7], [8], [9], [10] perform *digital* logic using the same physical devices that store binary information. By performing the logic exactly where the information is stored, data-transfer is effectively dwarfed [2]. These include works that exploit content-addressable-memories (CAMs) [8], [9], [10] to selectively apply write operations according to data stored in the memory (serving as inputs), and works that design logic gates from the underlying circuits connecting the memory devices [4], [5], [6], [7]. Several proposals for PIM architectures, such as memristive (Figure 1(a,b)) [4], [5], [11], [12], [13], [14] and DRAM-based (Figure 1(c,d)) [6], [7], [15], have essentially converged to a single abstract computational model that presents highly unique algorithmic capabilities (Figure 1(e)). First, consider the memory as a collection of m binary matrices, called *arrays*, each of dimension $r \times c$. A *bitwise* operation can be performed on columns of an array, and in parallel across all arrays, in a single cycle ($O(1)$ latency). For example, the bit-wise NOR [4] of any two columns can be computed and stored in a third column, all in a single cycle. This is possible as the logic is performed in a *distributed* fashion amongst the physical elements within arrays (with shared instructions), so there is no centralized computing unit that may cause a bottleneck. This attains massive parallelism for *bitwise* operations that bypass the memory interface.

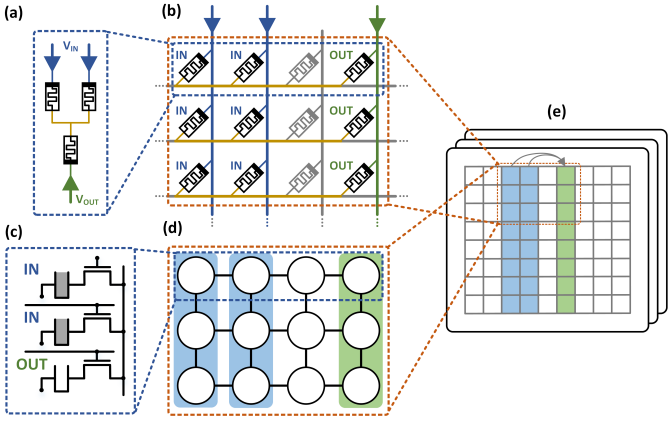

Fig. 1. Examples of PIM technologies for (a, b) memristive [4], [11], [13], [14] and (c, d) DRAM [15] memories. These follow (e) an abstract model of arbitrary bitwise column operations in $O(1)$ latency.

Expanding the massive bitwise parallelism to large-scale applications requires a strong foundation for fundamental arithmetic operations (addition, subtraction, multiplication, and division), for both fixed-point and floating-point numbers. While theoretically a functionally-complete set of logic gates can perform any function, the data layout plays a

• *O. Leitersdorf, D. Leitersdorf, J. Gal, M. Dahan, R. Ronen, and S. Kvatinsky are with the Technion – Israel Institute of Technology, Haifa, Israel. E-mail: orianl@campus.technion.ac.il; leitersdorf@cs.technion.ac.il; jonathan.gal@campus.technion.ac.il; mor.dahan@campus.technion.ac.il; ronny.ronen@technion.ac.il; shahar@ee.technion.ac.il.*

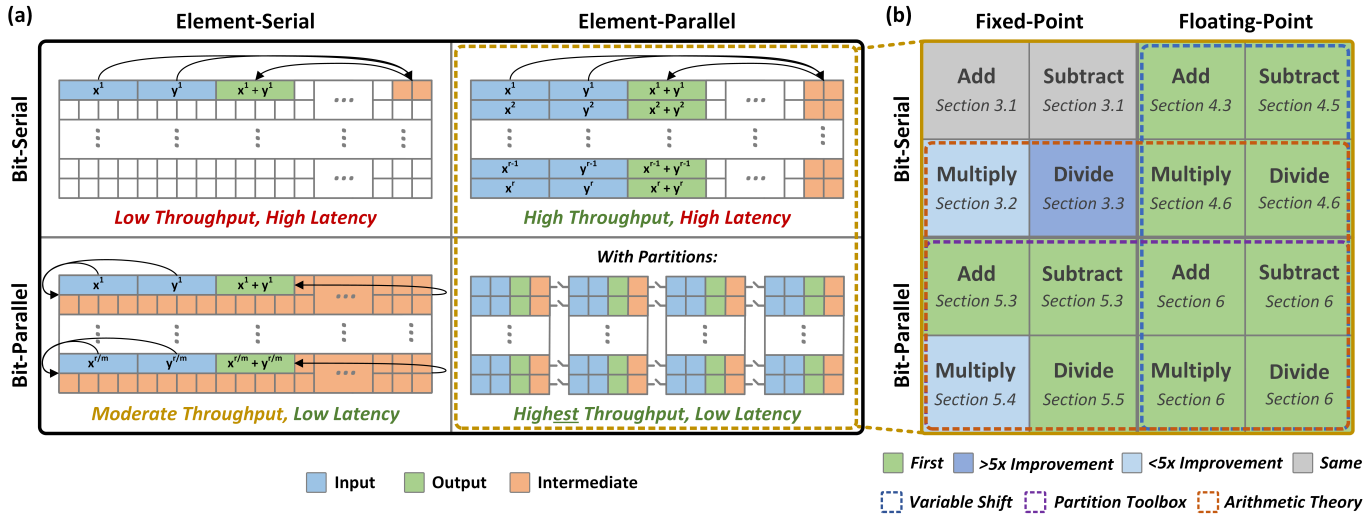

Fig. 2. (a) The various approaches to in-memory arithmetic developed in recent years. (b) The impact of this paper seen through the algorithms for the foundational arithmetic functions on both fixed-point and floating-point numbers, and with both bit-serial element-parallel and bit-parallel element-parallel approaches. The dashed rectangles highlight the three novel methods developed, and the algorithms which they effect.

crucial role in the efficient utilization of PIM. Figure 2(a) presents an overview of the approaches developed in recent years; we focus without loss of generality on a single array (as computation is in parallel across all arrays regardless). We describe the approaches through N-bit integer vector addition, where x^i refers to the i^{th} element in the vector:

1) *Bit-Serial Element-Serial:* Two inputs, x^1 and y^1, are stored within a single row of an array, and basic logic gates (e.g., NOR) *serially* construct an N-bit adder within that row (utilizing intermediate cells for temporary results). This has *low throughput* as only one addition is performed per array and *high latency* as the gates run serially; thus, this approach is *typically not used*. Note that column parallelism (constant-time column operations) is not utilized.
2) *Bit-Parallel Element-Serial [13], [16], [17], [18], [19]:* m rows are utilized as intermediate space to perform multiple parallel gates for the same N-bit adder by utilizing column parallelism, thereby enabling r/m adders per array. This provides *low latency* when the function possesses parallelism among the gates (typically best applicable to multiplication [17]), yet possesses *moderate throughput* as several rows perform a single addition. Furthermore, the relative area overhead can be high due to the intermediate cells [17], and the data-transfer between rows requires additional support for inter-row logic.
3) *Bit-Serial Element-Parallel [4], [5], [6], [7], [11], [13], [20], [21], [22], [23], [24]:* This approach performs the operations of the bit-serial element-serial approach in parallel across all rows – *with the exact same latency* – by exploiting column parallelism. That provides *high throughput* as r adders per array are performed simultaneously with identical latency; however, the latency remains rather high as gates are performed serially (a single gate per cycle per N-bit adder).
4) *Bit-Parallel Element-Parallel (parallel single-row) [25], [26]:* This recent approach gains both *higher throughput* and *low latency* by introducing *partitions* [11], [27] (see Section 5). The partitions dynamically divide the rows to enable multiple concurrent column operations. The adder is still performed within a single row (and in parallel across all rows), yet multiple gates are performed within each row concurrently. The potential drawback is that partitions may introduce additional physical overhead; however, a recent work has proposed a low-overhead design [27].

We focus on the *element-parallel* approaches as PIM is best suited for data-intensive applications which require high throughput. Figure 2(b) summarizes our contributions for 16 variants of arithmetic functions, establishing a state-of-the-art foundation for arithmetic in PIM. We propose the first known general-purpose digital PIM algorithm for a *majority* of the combinations (including cases previously considered impossible, such as floating-point addition [28]), while also presenting minor and major ($> 5\times$) improvements for others. We accomplish this via a combination of three methods:

- *Variable Shift:* We develop a novel algorithm for element-parallel *variable shifting:* each row i starts with numbers x^i and t^i, and the output is $x^i \ll t^i$. While this was previously considered *impossible* (as each row can have a different shift [28]), we attain this efficiently for the first time due to the combination of in-memory multiplexers and a logarithmic-shifter approach (without any custom periphery). We then tackle *variable normalization*: each row i starts with number x^i, and the output is x^i left-shifted until the MSB is one. This is far more difficult as the shift-amount is unknown, and yet we attain this with latency nearly identical to variable shift due to a technique inspired by a binary search.
- *Partition Toolbox:* We exploit a unique algorithmic topology enabled by partitions [27] towards an efficient *toolbox* of general-purpose routines. These include both generalizations of routines proposed in MultPIM [26], and two novel routines: *reduction* – reducing (e.g., AND, OR) bits of multiple partitions

to a single bit, and *prefixing* – each partition receives the reduction of bits in partitions before it.

- *Arithmetic Theory:* We provide a *new perspective* on historical, lesser-known, algorithms in computer arithmetic, demonstrating their effectiveness with element-parallel in-memory computing for the first time. For example, we utilize Karatsuba [29], [30] for bit-serial multiplication, parallel-prefix adders [30], [31] for bit-parallel addition, and carry-lookahead in division [30], [32], [33] for bit-parallel division. Interestingly, some of these algorithms are not effective in traditional systems [30], [33], yet unique considerations of PIM lead to their effectiveness here.

This paper is organized as follows. Section 2 provides background on a wide variety of PIM technologies and their compatibility with the abstract model. We start in Section 3 with bit-serial fixed-point arithmetic to establish the state-of-the-art approaches and our improvements, and continue in Section 4 with bit-serial floating-point arithmetic. Section 5 then shifts to the bit-parallel fixed-point approach, introducing bit-parallel addition/subtraction/division for the first time and improving bit-parallel multiplication. We combine bit-serial floating-point and bit-parallel fixed-point in Section 6 to establish bit-parallel floating-point algorithms. Section 7 evaluates AritPIM through a case study of memristive PIM implemented on a publicly-available cycle-accurate simulator, and Section 8 concludes this paper.

Throughout, we discuss abstract logic gates (e.g., AND, XOR, full-adder) and latency complexity (e.g., $O(N^2)$ where N is the representation size) for generality and concise explanations, with Section 7 reducing these to the underlying gates supported (e.g., NOR) and providing full implementations that prove correct results (e.g., matching the IEEE round-to-nearest ties-to-even *exactly*). We refer to *steps* rather than cycles, where each step performs a single abstract logic gate. Without loss of generality, as we focus on the element-parallel approaches, we often discuss gates performed within a *single-row*, as the generalization to all rows and arrays is the trivial repetition of the gates. For fixed-point, we discuss unsigned numbers (for simplicity) yet the algorithms can extended to signed. Lastly, v^i refers to the i^{th} element of vector v, x_i refers to bit i of number x (0 is the LSB), $x_{i:j}$ is bits i (inclusive) through j (exclusive), and $(x|y)$ concatenates x (higher bits) and y (lower bits).

2 Digital Processing-in-Memory (PIM)

We discuss two examples of digital PIM, memristive stateful-logic [4], [11], [12], [13] and DRAM [6], [7], [15], [28], as well as other PIM technologies. Overall, these include both architectures that are already commercially available, and emerging ones with vast potential that are backed by small experimental demonstrations. We show that *all* are compatible with the abstract model assumed in this paper.

2.1 Memristive Stateful-Logic

Memristive PIM with *stateful-logic* [2] utilizes an emerging physical device, the memristor, which inherently supports both storage and logic at the exact same place. Large-scale memristive memories (storage only) with high density are already commercially available (e.g., Intel Optane), and several studies [2], [12], [34] have experimentally demonstrated logic with memristors on a small-scale. Therefore, memristive digital PIM has the potential for an efficient large-scale practical implementation in the future.

Memristors are two-terminal resistive devices with a unique property: their resistance may be modified through an applied voltage. By dividing the range of possible resistance values to a binary classification (i.e., low resistance corresponds to logical one, high resistance corresponds to logical zero), a memristor can store a single bit through its resistance. The value of a memristor is read by applying a low voltage and measuring the current to derive the resistance, and their value may be written by applying a high voltage. Stateful-logic performs logical gates [2], [4], [11], [12], between the resistance states of memristors: for memristors arranged in the circuit shown in Figure 1(a), the final resistance state of the bottom memristor (output) depends on the resistance states of the top memristors (input) at the beginning of the operation due to the voltage-divider structure which is formed. An additional initialization cycle is required for the output prior to the gate operation [4].

Notably, memristive PIM may also be implemented through MRAM devices [13], [14], and possesses several favorable characteristics. Rather than dividing the resistance spectrum into ranges for each logical value that may be prone to errors due to noise, the MRAM device inherently has two stable states that correspond to low and high resistance. Furthermore, MRAM devices possess excellent endurance compared to other memristive technologies. MRAM PIM [13], [14] supports the same abstract model as Figure 1(a), and previous works [13], [14], [19], [24] have primarily explored the extension of MRAM computing to bit-parallel element-serial arithmetic.

Memristors are often arranged in dense crossbar-array structures. Such crossbars consist of vertical bitlines, horizontal wordlines, and memristors at the junctions. When considering the memristor as a binary storage element, a crossbar array essentially stores a binary matrix of information. The logic functionality of memristors is also compatible with the crossbar-array structure, as the same circuit observed in Figure 1(a) appears within a row of Figure 1(b). Interestingly, by applying voltages on bitlines, all of the rows perform the logic gate in parallel [4]. Essentially, this leads to a logic operation on columns of bits in a single cycle; e.g., the bit-wise NOR of two columns computed and stored in a third column, in a single cycle [4]. Therefore, the logic functionality within memristor crossbar arrays attains the abstract model presented in Figure 1(e) (see Section 1).

2.2 In-DRAM Logic

Recent works have demonstrated that Dynamic Random Access Memory (DRAM) technology (the leading form of computer memory today) can also support in-memory bit-wise logic computation by exploiting existing properties of DRAM cells [6], [7], [15], [28]. Various works have proposed minor changes (e.g., modifying decoders) to commercially-available DRAM memory in order to support in-memory logic gates [6], [15], [28], while other works utilized commercially-available DRAM *without modification* and

already have large-scale experimental demonstrations [7]. Therefore, in-DRAM logic has the potential for practical large-scale PIM implementation in the immediate future.

DRAM memory utilizes capacitors to store information through the stored charge. The range of possible charge levels in a single capacitor is divided into a binary classification, thereby storing a single bit per capacitor. To minimize the charge leakage from the capacitor, a single DRAM cell also includes a transistor connected in series. The state of a DRAM cell is read by discharging the capacitor and measuring the voltage using a sense amplifier, while the state is written by applying a voltage on the capacitor (thereby changing the stored charge). When three DRAM cells are connected as illustrated in Figure 1(c), then all three cells stabilize on a state which is majority of the states prior to the operation [6], [15]. Therefore, $\mathrm{Maj}(A,B,C) = AB + AC + BC$ is inherently supported by the DRAM cells. By setting one of the three cells to a known value before performing the gate, the OR and AND gates can be performed [15]. Inversion is supported with low-overhead modifications to sense amplifiers [15]. Note that the true inputs are typically first copied to these cells (from other cells using copy operations) due to the destructive-input property of the logic gates.

DRAM cells are often arranged in dense sub-array structures. Such structures consist of a grid of DRAM cells, as shown in Figure 1(d), with buffers and sense amplifiers to the left[1] of the array. We find that the circuit from Figure 1(c) also exists within every row of the sub-array in Figure 1(d), thereby enabling parallel bitwise logic-gate execution. Essentially, bit-wise operations on columns (e.g., majority, AND, OR, NOT) within sub-arrays can be performed in $O(1)$ cycles. Therefore, this attains the abstract model presented in Figure 1(e) (see Section 1).

2.3 Additional PIM Technologies

This section briefly mentions additional PIM technologies and discusses their compatibility with the abstract model.

2.3.1 SRAM

Computing via Static Random Access Memory (SRAM) technology performs logic operations using SRAM cells via the bitlines [23], [35] (similar to in-DRAM computing). This technique enables bit-wise operations on columns with constant time within SRAM arrays – supporting the abstract model. Further, a common form of in-SRAM processing involves associative computing [8], whereby the output is conditionally set according to a pattern search on the inputs. While such PIM architectures have already been fabricated [9], [10], the current limitation is that the memory size is limited by the low density of SRAM, thereby only supporting PIM for workloads with small datasets.

2.3.2 Non-Stateful Memristive

Memristive memory also supports non-stateful logic operations by performing the logic in the sense amplifiers of each crossbar (rather than using the memristors themselves). Pinatubo [36] is a leading example of such an architecture, providing bit-wise parallel logic operations that adhere to the abstract model. The drawback of non-stateful logic is the higher latency and energy consumption over stateful-logic.

1. Without loss of generality, we consider the sub-array transposed for terminology identical to memristive stateful-logic.

Algorithm 3.1 Bit-Serial Fixed-Point Addition

Input: N-bit x, N-bit y in a single row.
Output: $N+1$-bit result z in the same row, where $z = x + y$.

$carry \leftarrow 0$
for $i = 0, \ldots, N-1$ **do**
  Compute full-adder serially (e.g., serially executing 9 NOR gates and utilizing intermediate cells [4], [21])
  $z_i, carry \leftarrow \mathrm{FA}(x_i, y_i, carry)$
end for
$z_N \leftarrow carry$

2.3.3 FeFET

The Ferroelectric Field Effect Transistors (FeFET) [37] technology is emerging as a form of memory which can also inherently support basic logic gates [38]. Similar to Pinatubo [36], the logic is performed within the sense amplifiers and with parallelism identical to the abstract model. FeFET has the potential for highly compact and energy-efficient implementation in the future [37].

3 Bit-Serial Fixed-Point Arithmetic

This section details in-memory arithmetic algorithms for bit-serial computation on fixed-point numbers (the simplest case). We begin by introducing the state-of-the-art in-memory addition/subtraction algorithm [4], [13], [24], [26] which is based on a ripple-carry approach. We continue by introducing the state-of-the-art in-memory multiplication algorithm [20], based on the shift-and-add approach, which we then improve through Karatsuba [29], [30]. Finally, we address division by proposing a non-restoring algorithm which is optimized for element-parallel in-memory logic as it avoids the *conditional* subtraction efficiently.

3.1 Bit-Serial Fixed-Point Addition/Subtraction

We start with the simplest arithmetic functions, bit-serial fixed-point addition/subtraction, as an example of the format for in-memory arithmetic functions. These algorithms have been discussed in several previous PIM works [4], [13], [24], [26] as classic examples to bit-serial arithmetic. We discuss only addition as subtraction can be derived from two's-complement addition.

We begin with a formal description of the task. Assume that a single-row of an array contains two N-bit fixed-point numbers, x and y, and some additional *intermediate* cells that may be used freely. The algorithm may perform a single basic logic operation (e.g., NOR, full-adder) per *step*, with the inputs and outputs of the gate being cells in the single-row. The state of the row at algorithm completion should contain $z = x + y$ stored in a pre-determined range of cells, as an $N+1$-bit fixed-point number (including the carry-bit). Note that the algorithm *cannot read* the row at any time as this will not generalize to an element-parallel approach; rather, the algorithm must be based exclusively on *data-flow*.

Algorithm 3.1 details the ripple-carry approach to addition. The approach utilizes a single intermediate cell to store

Algorithm 3.2 Bit-Serial Fixed-Point Multiplication

Input: N-bit x, N-bit y in a single row.
Output: $2N$-bit result z in the same row, where $z = x * y$.
Base case using previous [20]. Note: $N_{thresh} \approx 20$.
if $N \leq N_{thresh}$ **then**
  $z \leftarrow (0 \cdots 0)_2$ {$2N$-bit.}
  for $i = 0, \ldots, N-1$ **do**
    Compute $p^i \leftarrow \text{AND}(x, y_i)$ *(serially over N bits).*
    for $j = 0, \ldots, N-1$ **do** $p^i_j \leftarrow \text{AND}(x_j, y_i)$
    Compute $z \leftarrow z + (p^i \ll i)$ *using Alg. 3.1.*
    $z_{i:i+N+1} \leftarrow z_{i:i+N} + p^i$
  end for
Proposed Karatsuba recursion.
else
  Compute using recursive calls and Alg. 3.1.[2]
  $t'_1 \leftarrow (x_{0:N/2} + x_{N/2:N}) * (y_{0:N/2} + y_{N/2:N})$
  $t_0 \leftarrow x_{0:N/2} * y_{0:N/2}, t_2 \leftarrow x_{N/2:N} * y_{N/2:N}$
  $t_1 \leftarrow t'_1 - t_0 - t_2$
  Compute $z \leftarrow t_0 + t_1 \ll N/2 + t_2 \ll N$ *using Alg. 3.1.*
  $z \leftarrow (t_2|t_0)$
  $z_{N/2:2N} \leftarrow z_{N/2:2N} + t_1$
end if

the current carry between the bits, and iteratively computes the full-adders from the LSB to the MSB. Notice that this differs from a traditional ripple-carry adder in that all gates (e.g., NOR) are applied serially to the rows of the memory (e.g., requiring cells in each row that store intermediate results). Overall, the latency is $O(N)$ steps. This general approach is optimized for different PIM technologies in recent works (e.g., CRAM [13], FELIX [11] and MultPIM [26]). Such optimizations include efficient constructions of the full-adder from the logic gates supported by memristive PIM, and the storage of both the carry and the NOT carry throughout the iterations to save an additional cycle.

3.2 Bit-Serial Fixed-Point Multiplication

We begin this section by introducing the state-of-the-art approach introduced by Haj-Ali *et al.* [20], and continue by proposing an improvement based on the Karatsuba [29], [30] recursion; while Karatsuba is typically only effective for extremely-wide numbers [30] (e.g., thousands of digits), unique considerations for digital PIM enable improvement for regularly-sized numbers (e.g., 32-bit). The overall task is: the row begins with N-bit fixed-point numbers x and y, and contains the $2N$-bit result $z = x * y$ after completion.

The shift-and-add approach to multiplication constructs

$$x * y = \sum_{i=0}^{N-1} p^i \ll i, \quad \text{where } p^i = \text{AND}(x, y_i). \tag{1}$$

This approach first initializes the $2N$-bit output z to zero, and then iteratively adds $p^i \ll i$ to z for each i. Only an N-bit adder is required due to the zeros contained in $p^i \ll i$ (only N bits are non-zero) and z (top $N - i$ bits are zero during the i^{th} iteration). The base case in Algorithm 3.2 presents the state-of-the-art PIM algorithm [20] which is based on this shift-and-add. The algorithm iterates over each i, computing the i^{th} partial product, p^i, and adding it to the current sum z. The shift is not computed explicitly, rather the elements in z are merely accessed shifted (the N columns chosen from the $2N$ are different in each iteration); that is, the shift is *simulated*. Overall, the latency is $O(N^2)$.

The Karatsuba [29], [30] approach reduces asymptotic complexity to $O(N^{\log_2 3}) \approx O(N^{1.58})$ through an optimization to the recursive expression for N-bit multiplication. Consider $x = (x_1|x_0)$, $y = (y_1|y_0)$, the separation of each N-bit number to the upper and lower bits ($N/2$-bit numbers). The naive recursion for multiplication notes that,

$$x * y = \underbrace{x_0 y_0}_{t_0} + \underbrace{(x_0 y_1 + x_1 y_0)}_{t_1} \ll N/2 + \underbrace{x_1 y_1}_{t_2} \ll N. \tag{2}$$

Thus, in the naive recursion, an N-bit multiplication requires *four* $N/2$-bit multiplications. The Karatsuba approach reduces the number of $N/2$-bit multiplications to *three* by computing t_1 with a single $N/2$-bit multiplication,

$$t_1 = (x_1 + x_0)(y_1 + y_0) - t_2 - t_0. \tag{3}$$

In traditional computing systems, this approach is only used for large-number multiplication [30] as the objective is to minimize the critical path and bit-level access is not possible. Conversely, for *bit-serial* in-memory computing, the latency is the total number of gates and arbitrary bit-level operations may be executed. Therefore, we utilize (3) with both the shift-and-add approach (base case) and Algorithm 3.1 to propose Algorithm 3.2. The algorithm contains a base case of performing shift-and-add directly if N is small, and otherwise performs the three recursive calls and computes the output according to (2) and (3). Overall, the latency is $O(N^{1.58})$, providing minor (yet significant) improvements starting[3] at approximately $N \approx 20$.

3.3 Bit-Serial Fixed-Point Division

Here, we tackle the most complex operation out of the four elementary arithmetic operations: division. We begin with background on restoring and non-restoring division as theoretical concepts, and then continue by presenting a novel algorithm for in-memory division that is based on a customized non-restoring approach. While bit-serial division does exist [39] (restoring), our proposed algorithm is based on a different approach (non-restoring) that is better suited for in-memory computing as it inherently avoids the *conditional* subtraction. Such conditional operations (i.e., branches) are not directly compatible with the abstract model which requires that all rows operate in lockstep, thus they are converted to a sequence of operations that serially evaluate both branch outcomes and then select the output with a multiplexer. The overall task is defined as the integer[4] division of $2N$-bit dividend z by N-bit divisor d, with N-bit quotient q and N-bit remainder r (i.e., $z = qd + r$ for $r < d$).

2. By calculating t'_1 before t_0 and t_2, we can reuse the cells that stored $x_{0:N/2} + x_{N/2:N}$ and $y_{0:N/2} + y_{N/2:N}$ for storing t_0 and t_2 (thereby reducing the proposed algorithm's space complexity).

3. The crossover point N_{thresh} depends on the latency for a recursive Karatsuba step (additions/subtractions and the smaller multiplications) compared to that of naive shift-and-add. The value $N_{thresh} \approx 20$ is found by increasing N_{thresh} until Karatsuba reduces the overall latency. This crossover is largely independent of the assumed logic gates (e.g., NOR) since the compared latencies (Karatsuba step and shift-and-add) are both *primarily* proportional to the number of cycles per 1-bit full-adder; regardless, the exact value may vary slightly.

4. General fixed-point can be derived from such integer division.

Algorithm 3.3 Non-Restoring Division *(Theoretical)*

Input: $2N$-bit dividend z, N-bit divisor d
Output: N-bit quotient q, N-bit remainder r, where $z = qd + r$ and $r < d$

1: $q \leftarrow 2^{N-1}, r \leftarrow z_{N-1:2N}$
2: **for** $i = N-1, \ldots, 0$ **do**
*Add/subtract conditional on **previous** bit from q.*
3: **if** q_i **then** $r \leftarrow r - d$ **else** $r \leftarrow r + d$
4: $q_{i-1} \leftarrow (r \geq 0)$
5: $r \leftarrow 2 \cdot r + z_{i-1}$
6: **end for**
Non-restoring representation corrections.
7: $q \leftarrow 2 \cdot q + 1$
8: **if** $r < 0$ **then** $q \leftarrow q - 1, r \leftarrow r + d$

Algorithm 3.4 Bit-Serial Fixed-Point Division

Input: $2N$-bit dividend z, N-bit divisor d in a single row.
Output: N-bit quotient q, N-bit remainder r, in the same row, where $z = qd + r$ and $r < d$.

1: $q_N \leftarrow 1, r \leftarrow z_{N-1:2N}$
2: **for** $i = N-1, \ldots, 0$ **do**
Compute using vectored-XOR (serially over the bits of d with q_{i+1}) and Alg. 3.1, with q_{i+1} as carry-in.
3: $(q_i | r_{1:N}) \leftarrow r + \mathrm{XOR}(d, q_{i+1}) + q_{i+1}$
4: $r_0 \leftarrow z_{i-1}$
5: **end for**
6: $q_0 \leftarrow r'_{N-1}$ {Already shifted.}
Compute using Alg. 3.1.
7: $r \leftarrow r + \mathrm{AND}(d, r_{N-1})$

Restoring division [30] is based on *conditionally* subtracting the divisor from the current remainder, as long as the result remains non-negative. Formally, $r^{j+1} = 2 \cdot r^j - q_{j+1} \cdot d$, where r^j is the j^{th} partial remainder; restoring division chooses $q_{j+1} \in \{0, 1\}$ such that $0 \leq r^{j+1} < d$, implying $q_{j+1} = 1$ if $2r^j - d \geq 0$, else $q_{j+1} = 0$. Thus, overall it iterates over: shifting the remainder (computing $2 \cdot r^j$), and subtracting d from $2 \cdot r^j$ but *only if* it remains non-negative.

Non-restoring division [30] enables intermediate negative remainders to avoid the conditional subtraction. Instead of conditional subtraction, it either subtracts or adds (the benefit is that the choice is known at the end of the *previous* iteration). On a theoretical level, $q_i \in \{-1, +1\}$ rather than $q_i \in \{0, 1\}$. Practically, $-1, +1$ are stored as $0, 1$, respectively, and corrections are performed at the end. Algorithm 3.3 presents the *theoretical* algorithm for binary non-restoring division, slightly modified with PIM in mind.

Algorithm 3.4 details the proposed bit-serial fixed-point divider, based on the theoretical approach from Algorithm 3.3. Specifically, we utilize these optimizations:

1) *Conditional Addition/Subtraction:* Control-flow for conditional addition/subtraction (Alg. 3.3, Line 3) is replaced with data-flow. Similar to [39], we could compute $r - d$ and $r + d$ (serially), and implement a multiplexer to choose according to q_i; however, this results in large overhead for the multiplexer and computation of both $r - d$ and $r + d$. Instead, we utilize properties of the two's-complement representation by performing bit-wise exclusive-or of d with q_i, and then performing addition between the result, r, and a carry-in of q_i (Alg. 3.4, Line 3) [30].[5]
2) *Quotient-Bit Update:* Updating the quotient bit (Alg. 3.3, Line 4) is achieved by checking the most-significant-bit of the remainder (two's complement). This is performed as part of the addition (Alg. 3.4, Line 3), without additional steps.
3) *Remainder Shifting:* Shifting the remainder (Alg. 3.3, Line 5) is *simulated* by the algorithm as the addition (Alg. 3.4, Line 3) already stores the shifted result.
4) *Non-Restoring Correction:* Correcting the representation mismatch (Alg. 3.3, Lines 7, 8) is converted to data-flow as follows. $2 \cdot q$ is replaced by a shift of q that is *simulated* throughout previous references to q_i (Alg. 3.4, Lines 1-5). The $+1$ (Alg. 3.3, Line 7), and the subsequent conditional -1 (Alg. 3.3, Line 8) if $r < 0$, are replaced with $q_0 \leftarrow (r < 0)' = r'_{N-1}$ (Alg. 3.4, Line 6). The conditional $r \leftarrow r + d$ if $r < 0$ is replaced with the addition of the bitwise-and of d and r_{N-1} to r (Alg. 3.4, Line 7).

5. Note that this optimization is not possible with restoring division (e.g., with AND) as the sign of $r - d$ must be computed regardless.

Overall, the latency of the proposed algorithm is $O(N^2)$ steps, with the constant significantly improved over the previous state-of-the-art restoring division [39] primarily due to the non-restoring approach being better suited for PIM. Notice that a Karatsuba-style approach for division [40] does not improve the *worst-case* complexity, and thus is not applicable to in-memory computing (as requiring purely data-flow implies that latency is limited by the worst case).

4 Bit-Serial Floating-Point Arithmetic

This section expands the proposed bit-serial fixed-point arithmetic algorithms to floating-point numbers, thereby supporting the numerous applications that require floating-point accuracy. In-memory floating-point addition *was previously considered impossible* as the shift operations in the alignment and normalization steps inherently require control-flow that is difficult to perform in an element-parallel manner [28]. Conversely, we attain an in-memory unsigned floating-point addition algorithm *for the first time* by first proposing a simple (but powerful and efficient) element-parallel variable shift routine. This routine receives numbers x (N_x-bit) and t (N_t-bit) in each array row, and outputs $x \gg t$ in that row (where the shift amount may be *different for each row*). Furthermore, we generalize the proposed variable shifter to a variable normalization routine (left-shifts each number until the MSB is one) through a technique inspired by a *binary search*, thereby also supporting *signed* floating-point numbers. Lastly, we demonstrate floating-point multiplication/division by utilizing their fixed-point counterparts and the variable shift routine with $N_t = 1$.

4.1 Floating-Point Representation

This section provides background on the floating-point representation [30]. The representation is essentially inspired by the scientific number format, thereby representing a wide range of *real* numbers (e.g., from 2^{-127} to 2^{127}). Thus, the representation is crucial for large-scale applications that process real numbers. We consider the IEEE 754 format,

which includes a sign-bit s (1-bit), an unsigned exponent e (N_e-bit), and an unsigned mantissa m (N_m-bit). The value of the number x is defined as

$$x = (-1)^s \cdot 2^{e-b} \cdot (1.m), \tag{4}$$

where b (bias) is a constant value (e.g., 127 for 32-bit), and the "1." is known as the *hidden-bit*. We adhere to the IEEE standard of *round to nearest, ties to even*. For simplicity, we do not consider NaN/Inf/subnormals/overflows, yet the proposed algorithms could be modified to address these rare cases (if required by the application).

Arithmetic with floating-point numbers can be significantly more complex than fixed-point numbers due to the alignment and normalization steps, specifically with *addition* and *subtraction*. Counter-intuitively, multiplication and division algorithms for floating-point numbers are rather simple generalizations from fixed-point algorithms. Consider the multiplication of two floating-point numbers, x_1 and x_2,

$$\begin{aligned} x_1 * x_2 &= (-1)^{s_1} \cdot 2^{e_1-b} \cdot (1.m_1) \cdot (-1)^{s_2} \cdot 2^{e_2-b} \cdot (1.m_2) \\ &= (-1)^{s_1 \oplus s_2} \cdot 2^{(e_1+e_2-b)-b} \cdot ((1.m_1) \cdot (1.m_2)). \end{aligned} \tag{5}$$

Thus, floating-point multiplication can be reduced to exclusive-or (XOR) for the sign-bit, addition for the exponent, and then fixed-point multiplication for the mantissas. Note that if the fixed-point multiplication results in a mantissa in the range $[2, 4)$, then the mantissa is right-shifted once and the exponent is incremented [30]. Floating-point division can be similarly reduced to the XOR of the sign-bits, the subtraction of the exponents, and then fixed-point division of the mantissas (with a single conditional left-shift and exponent decrement if the mantissa is in $[0.5, 1)$) [30].

Conversely, unsigned floating-point *addition* is significantly more complex as the input numbers must be aligned to match exponents before the addition (e.g., $1.2 \cdot 10^8 + 4.0 \cdot 10^7 = 1.2 \cdot 10^8 + 0.40 \cdot 10^8 = 1.60 \cdot 10^8$ for base 10). Such alignment can lead to complex control-flow mechanisms. Furthermore, addition may also require an additional *single* conditional right-shift normalization (e.g., $7.2 \cdot 10^2 + 4.1 \cdot 10^2 = 11.3 \cdot 10^2 = 1.13 \cdot 10^3$). Overall, the algorithm follows these steps: (1) computing the difference of the exponents, (2) shifting the mantissa of the number with the smaller exponent to match the number with the larger exponent, (3) adding the mantissas, and (4) normalizing the result.

Interestingly, signed addition, from which subtraction can be derived, requires a more complex normalization that left-shifts the mantissa several times until the MSB is one (e.g., $(1.0013 \cdot 10^1) + (-1.0000 \cdot 10^1) = 0.0013 \cdot 10^1 = 1.3 \cdot 10^{-2}$) [30]. Therefore, steps (1), (2), (3), remain similar, while step (4) requires more complex mechanisms.

4.2 Bit-Serial Variable Shift Routine

This section proposes the first in-memory algorithm for variable shifting through a simple-but-powerful approach that utilizes *simulated* in-memory multiplexers and a logarithmic-shifter approach (requiring no custom periphery at all). The task is defined as follows: the row begins with N_x-bit *integer* x and N_t-bit *integer* t, and the output is $z = x \gg t$[6]. Note that the shift amount is variable (may be different in each row in the memory array), and recall that the algorithm must be based exclusively on data-flow.

6. Left-shift is also supported due to symmetry.

Algorithm 4.1 Bit-Serial Variable Shift Routine

Input: N_x-bit x, N_t-bit t, in a single row.
Output: N_x-bit result z in the same row, where $z = x \gg t$.

```
z ← x
for j = 0, ..., min(N_t − 1, log2(N_x) − 1) do
    Compute z ← mux_{t_j}(z ≫ 2^j, z) as follows:
    for i = 0, ..., N_x − 2^j − 1 do z_i ← mux_{t_j}(z_{i+2^j}, z_i)
    for i = N_x − 2^j, ... N_x − 1 do z_i ← AND(¬ t_j, z_i)
end for
```

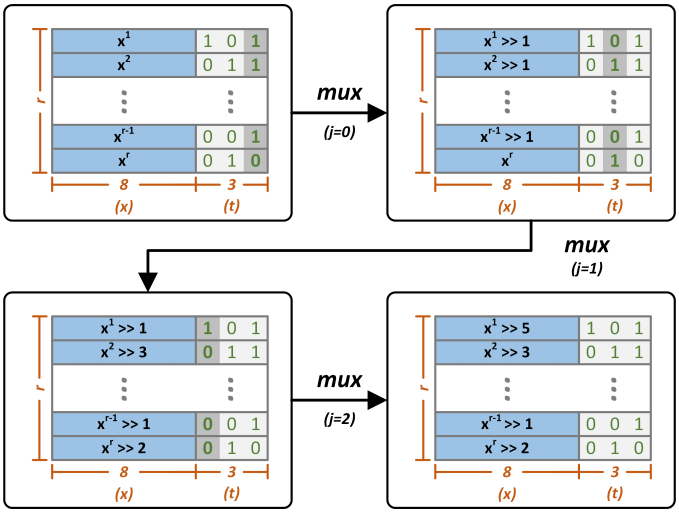

Fig. 3. An example execution of the proposed in-memory bit-serial element-parallel *variable shift* algorithm with $N_x = 8$ and $N_t = 3$. Each row i right-shifts x^i by its corresponding t^i through $\log(N_x)$ iterations that construct the shift via multiplexers from shifts of size $2^0, 2^1, \cdots$.

We first address the special case of $N_t = 1$. In this case, $z = x \gg 1$ if $t_0 = 1$ and $z = x$ otherwise ($t_0 = 0$); that is,

$$z = \text{mux}_{t_0}(x \gg 1, x) = \text{mux}_{t_0}(x_{1:N_x}, x_{0:N_x}). \tag{6}$$

where $x_{1:N_x}$ is zero-extended with an additional bit (at the MSB). We note that a $2:1$ N_x-bit multiplexer can be derived from basic logic gates (e.g., NOR), thereby enabling the implementation of an *in-memory multiplexer* algorithm [39] (not a dedicated multiplexer circuit, rather a bit-serial algorithm derived from a sequence of gates). Therefore, we find that the case of $N_t = 1$ can be addressed with a single $2:1$ N_x-bit multiplexer algorithm, exclusively through data-flow.

We efficiently extend the special case of $N_t = 1$ to any N_t through a *logarithmic-shifter* approach. The basic concept of a logarithmic-shifter can be seen in the following example: shifting x by 11 is identical to shifting x by 1, then by 2, and then by 8 (the binary representation of 11). Therefore, the proposed algorithm begins with $z = x$ and then performs $\log_2(N_x)$ iterations[7] where the j^{th} iteration executes:

$$z = \text{mux}_{t_j}(z \gg 2^j, z) = \text{mux}_{t_j}(z_{2^j:N_x}, z_{0:N_x}). \tag{7}$$

Algorithm 4.1 details the overall variable shift algorithm, and Figure 3 illustrates an example execution across all rows in an array (in parallel). The overall latency is

7. $\lceil \log(N_x) \rceil$ is computed as part of the algorithm compilation (i.e., by the controller as N_x is constant).

Algorithm 4.2 Bit-Serial Floating-Point Addition (Unsigned)

Input: Unsigned floating-point x, y (N_e-bit exponents $x.e, y.e$, N_m-bit mantissas $x.m, y.m$) in a single row.
Output: Unsigned floating-point z (N_e-bit exponent $z.e$, N_m-bit mantissa $z.m$) in the same row, where $z = x + y$.

Exponent difference using Alg. 3.1:
1: $\Delta e \leftarrow x.e - y.e$.
2: $z.e \leftarrow \text{mux}_{\Delta e \geq 0}(x.e, y.e)$ {Maximum.}
Conditional swap using vectored mux:
3: $x'.m \leftarrow \text{mux}_{\Delta e \geq 0}(x.m, y.m)$
4: $y'.m \leftarrow \text{mux}_{\Delta e \geq 0}(y.m, x.m)$
Alignment using Alg. 4.1:
5: $y'.m \leftarrow y'.m \gg |\Delta e|$
Integer addition using Alg. 3.1:
6: $z.m, carry \leftarrow x'.m + y'.m$
Normalization using Alg. 4.1 and Alg. 3.1 (with $N_t = 1$):
7: $z.m \leftarrow z.m \gg carry$
8: $z.e \leftarrow z.e + carry$

$O(N_x \log(N_x))$ steps. Note that the zero-extension of $z_{2^j:N_x}$ for the upper 2^j bits is replaced by performing an AND gate rather than $2:1$ 1-bit multiplexer at those indices.

4.3 Bit-Serial Floating-Point Unsigned Addition

This section utilizes the novel variable-shift algorithm to propose the first in-memory floating-point addition algorithm for unsigned[8] numbers. Algorithm 4.2 details the overall algorithm, essentially performing the theoretical steps for floating-point addition (see Section 4.1), while utilizing the variable-shift routine (Algorithm 4.1) for the alignment and single-bit normalization. Absolute value ($|\Delta e|$) is implemented by computing the exclusive-or of Δe and $\Delta e \geq 0$ (MSB), and then adding $\Delta e \geq 0$ using Algorithm 3.1. The *hidden-bit* is addressed by concatenating a 1 to the mantissas and IEEE *rounding* is addressed using sticky/round/guard bits [30] (not shown); details are available in the code repository. We utilize an in-memory conditional-swap (in-memory multiplexers) to guarantee that the exponent of x' is at least that of y'. The overall latency is $O(N_m \log(N_m) + N_e)$ steps.

4.4 Bit-Serial Variable Normalization Routine

To support signed floating-point addition/subtraction as well, we require an additional routine that is significantly more complex than variable shift: the input is x in each row, and the output is *both* x left-shifted until the MSB is one and the shift amount stored as a number in the same row. The difficulty arises from the fact that the shift amount is not known in advance (and still may be different in each row). Naive attempts to compute the shift amount and then perform the variable shift routine (e.g., by using full-adders to count the number of leading zeros) will lead to a massive overhead in latency (approximately 300% more steps). Conversely, inspired by a binary search, we propose a minor modification to the variable shift routine that adds a small number of steps (approximately 7%) and solves the variable normalization task exactly.

We begin by describing the approach through an example. Consider $N_x = 8$, with $x = (00000110)$; thus, the *desired output* is $z = (11000000)$ and $t = 5 = (101)$. The algorithm proceeds as follows. As the highest $N_x/2 = 4$ bits of x are all zero (equivalently, their OR is zero), then $t_2 = 1$ (that is, $t \geq 4$) and thus we already set $z \leftarrow x \ll 4 = (01100000)$. As the highest $N_x/4 = 2$ bits of z are *not* all zero (equivalently, their OR is non-zero), then $t_1 = 0$ and z is not changed. As the highest $N_x/8 = 1$ bits of z are all zero, then $t_0 = 1$ and $z \leftarrow z \ll 1 = (11000000)$. Therefore, the algorithm correctly returned $z = (11000000)$ and $t = (101) = 5$. In general, for each $j = \log_2(N) - 1, \ldots, 0$, the algorithm performs:

$$t_j \leftarrow \neg(z_{N_x - 2^j} \vee \cdots \vee z_{N_x - 1}), \quad z \leftarrow \text{mux}_{t_j}(z \ll 2^j, z) \quad (8)$$

The above procedure is analogous to a binary search on the OR-prefix of x (searching for the first 1 in the word): at each iteration, the size of the current search window (the interval which contains the first 1) is decreased two-fold, and the choice between the left/right intervals is performed by the in-memory multiplexer (always "pulling" the chosen interval to the left). Algorithm 4.3 details the overall proposed normalize-shift algorithm, which modifies the variable-shift algorithm by including the computation of t_j within each iteration. Interestingly, this integrates the computation of t

Algorithm 4.3 Bit-Serial Variable Normalization Routine

Input: N_x-bit x in a single row.
Output: N_x-bit z, $\log_2(N_x)$-bit t, such that the number of leading zeros in x is t, and $z = x \ll t$.
1: $z \leftarrow x$
2: **for** $j = \log_2(N_x) - 1, \ldots, 0$ **do**
Compute $t_j \leftarrow \neg(z_{N_x - 2^j} \vee \cdots \vee z_{N_x - 1})$.
3: $temp \leftarrow 0$
4: **for** $i = N_x - 2^j, \ldots N_x - 1$ **do**
5: $temp \leftarrow temp \vee z_i$
6: **end for**
7: $t_j \leftarrow \neg\, temp$
Compute $z \leftarrow \text{mux}_{t_j}(z \ll 2^j, z)$.
8: Omitted as nearly identical to Lines 3, 4 of Alg. 4.1.
9: **end for**

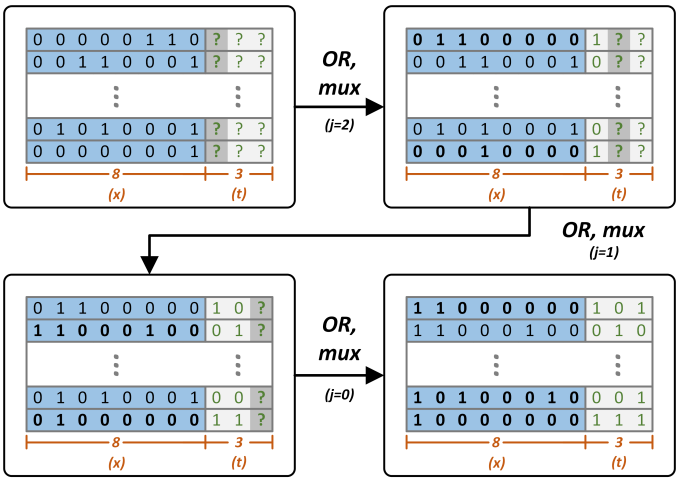

Fig. 4. An example execution of the proposed in-memory bit-serial element-parallel *variable normalization* algorithm with $N_x = 8$. Each row i left-shifts x^i until the MSB is one, while also producing the shift amount t^i, via shift iterations that consist of a multiplexer and OR reduction.

8. This algorithm is only applicable when both numbers possess the same sign. See Section 4.5 for an extension to signed addition.

within the variable-shift iterations. The overall latency is

$$O\left(\sum_{j=0}^{\log_2 N_x - 1} (N_x + 2^j)\right) = O(N_x \log(N_x) + N_x); \quad (9)$$

that is, the complexity remains identical to the $O(N_x \log(N_x))$ from variable-shift even though the task is significantly more difficult (since shift is unknown). Notice that the overhead over variable-shift is only the OR computations for t, which is $O(\sum_{j=0}^{\log_2 N_x - 1} 2^j) = O(N_x)$ steps total, leading to the very low increase in latency over variable-shift (approximately 7% for $N_x = 24$).

4.5 Bit-Serial Floating-Point Signed Addition

This section utilizes the novel bit-serial variable normalization to propose the first in-memory *signed* floating-point addition. Subtraction can be derived from such *signed* addition by simply inverting the sign-bit of the second input. The proposed algorithm extends Algorithm 4.2 as follows: $x'.m$ is replaced with $-x'.m$ if $x.s \neq y.s$ [30] (performed using data-flow through a technique similar to the Section 3.3), and Algorithm 4.3 is utilized at the end to left-normalize $z.m$ (in addition to the simpler single right-shift normalization using Algorithm 4.1). The sign of the overall output is computed through data-flow via an expression involving $\Delta s = \text{XOR}(x.s, y.s)$, whether $z.m$ was negative, and whether x, y were swapped; details are available in the code repository. Overall, this algorithm possesses the same properties as the unsigned addition algorithm: the abstract PIM model is utilized without modifications (i.e., no custom periphery required), and the algorithm operates within a single-row via data-flow (enabling element-parallel execution). The overall latency remains $O(N_m \log(N_m) + N_e)$ steps, yet the constant increases slightly (over unsigned) as the variable-normalization (Algorithm 4.3) is performed in addition to the variable-shifting (Algorithm 4.1).

4.6 Bit-Serial Floating-Point Multiplication/Division

As detailed in Section 4.1, these algorithms are derived naively from the corresponding bit-serial fixed-point algorithms to compute the result mantissa, and fixed-point addition/subtraction to compute the result exponent. The variable shift routine with $N_t = 1$ is used for the final single-step normalization of the mantissa. The overall latency is $O({N_m}^{\log_2(3)} + N_e) \approx O({N_m}^{1.58} + N_e)$ steps for multiplication and $O({N_m}^2 + N_e)$ steps for division.

4.7 Related In-Memory Floating-Point Works

The difficulty of in-memory element-parallel floating-point operations (e.g., variable shifting [28]) has led to little research on the subject. For example, DRISA [28] explicitly mentions the lack of a variable-shift routine obstructing the support for floating-point operations. Nonetheless, a few research works [10], [22], [41] have previously attempted such floating-point operations. Yet, their algorithms require content-addressable-memory (CAM) functionality to be integrated within the arrays (no longer adhering to the abstract model). Specifically, they require the existence of a *search* operation which can select certain rows for a mask based on the data stored in row, and then only apply column operations according to the mask. These search operations are exploited towards variable shift by iterating over all possible shift quantities (all values of t) and shifting only the rows which correspond to exactly that shift amount.

Unfortunately, the integration of a CAM within memristive crossbar arrays may increase the memory area by approximately $12.5\times$ [42]. This overhead has directly led to the algorithms *not* being widely adopted [5], [39]; further, CAMs may not be compatible with other forms of PIM (e.g., DRAM). *Conversely, our proposed bit-serial floating-point algorithms require no modifications, are compatible with many additional forms of PIM (e.g., DRAM), and, even without resorting to extra hardware, are faster than the previous works [10], [22], [41] due to the logarithmic shifter and binary-search approaches.*

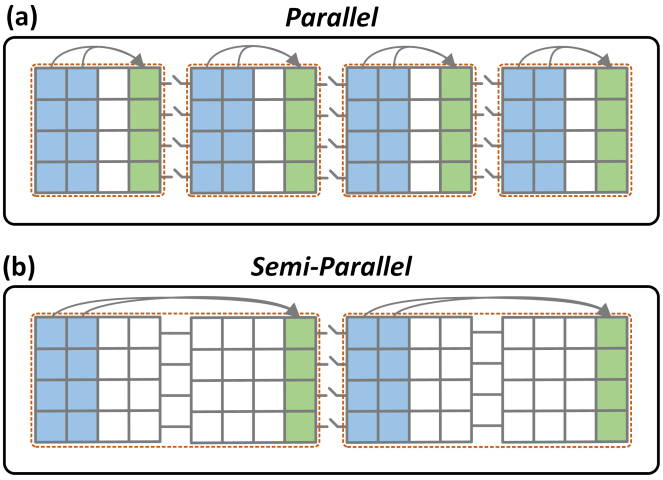

Fig. 5. Partitions emerge when switches partition arrays to increase parallelism, for (a) fully-parallel (all switches disconnected) and (b) semi-parallel operations (some switches disconnected).

5 Bit-Parallel Fixed-Point Arithmetic

We overcome an inherent limitation of the bit-serial element-parallel approaches by utilizing an emerging technique of *partitions* [11], [27], thereby introducing a highly-unique computation model that we exploit for bit-parallel element-parallel arithmetic. While the bit-serial element-parallel approach provides high throughput from the parallel computation across all rows and all arrays, the gates within each row are performed serially (e.g., one NOR at a time). This leads to high latency and low utilization of the potential computational power of PIM: all the $r \cdot c$ devices of each array have inherent logic capabilities, yet we only activate $O(r)$ devices per cycle. Partitions have recently emerged as a minor modification to PIM architectures that overcomes this limitation by enabling multiple concurrent column operations within a single array. Every array is horizontally divided into k partitions, each sized $r \times (c/k)$, which are connected via $k - 1$ sets of r switches. This enables concurrent execution in different partitions when the switches are disconnected, and the ability to efficiently transfer data between partitions (using column operations) when some switches are connected through *semi-parallel* operations, as illustrated in Figure 5. We exploit this unique computational

model towards a suite of *bit-parallel element-parallel* fixed-point arithmetic algorithms that vastly improves latency.

This section begins by providing further details on the computational model of partitions, and the support for partitions in PIM architectures. We then develop a general-purpose *toolbox* for techniques that efficiently exploit the unique computational model of partitions. We first utilize this toolbox for fast fixed-point addition and subtraction through the parallel-prefix addition concept [30], [31]. We then present MultPIM [26], the state-of-the-art for bit-parallel multiplication, and improve it by utilizing the proposed bit-parallel addition algorithm. Bit-parallel division is more complex than multiplication due to the conditional operations, and thus we utilize a lesser-known historical concept of carry-lookahead in division [30], [32], [33] alongside a novel in-memory carry-lookahead routine.

5.1 Partitions

Partitions have recently emerged [11], [27] as a simple modification to PIM architectures that vastly improves parallelism with a highly-unique computational model. The dynamically-controlled switches dividing the partitions in each array allow merging adjacent partitions to perform gates amongst data stored in different partitions. At the extreme cases, we find that disconnecting all switches (parallel) provides maximal parallelism but with minimal flexibility (each gate is constrained to a single partition), and connecting all switches (serial) provides minimal parallelism but with maximal flexibility (each gate can access all of the columns of the array). Semi-parallelism refers to the case where only some switches are connected, providing intermediate parallelism and flexibility, see Figure 5.

The implementation of partitions has been discussed in the context of memristive PIM using transistors [11], [27], and should be applicable to DRAM PIM as well. Furthermore, partitions nearly attaining the model assumed in this paper are already included in a commercially-available SRAM processor [9]. Counter-intuitively, the overhead required for the switches is rather minuscule in comparison to the array size [11], [27], with the more significant overhead being the periphery and control required to select all of the involved columns [27]. Therefore, PartitionPIM [27] proposes a reduced set of semi-parallel operations that requires various patterns to significantly reduce such overhead; in AritPIM, we assume this reduced set (minimal model [27]).

Partitions in arithmetic accelerate element-parallel algorithms by enabling multiple concurrent logic gates per function, see Figure 2(a). We assume in this section that $k = N$ (representation size is exactly the number of partitions)[9]. In this case, the inputs and outputs are stored in a *strided* format: e.g., 32-bit integers are stored with a single bit per partition. This enables $O(1)$ latency for bitwise integer operations (e.g., integer OR of two 32-bit numbers) as all partitions operate in parallel. Arithmetic functions more complex than bitwise integer operations require semi-parallelism to share information between the different partitions. Only few works have explored bit-parallel arithmetic [25], [26].

9. The proposed algorithms can be trivially extended to any $k > N$, and may also be generalized for the case of $k < N$.

5.2 Partition Toolbox

We present a broad-range of partition techniques, constituting a general-purpose toolbox for bit-parallel algorithms. We start by briefly explaining and generalizing the *broadcast* and *shift* techniques from MultPIM [26], and we then propose the novel *reduction* and *prefix* techniques. Prefix is complex as it is intuitively serial; yet, inspired by Brent-Kung [31], we propose an efficient parallel algorithm with logarithmic time. For the remainder of the text, k denotes the total number of partitions, p_i denotes the i^{th} partition, and $p_i.x$ refers to bit x_i of number x (which is stored in p_i). We also assume an abstract single-input *identity* gate $\mathrm{Id}(A) = A$ for simplicity (e.g., implemented using two NOT gates).

5.2.1 Shift

Each partition stores a bit, and we shift the bits among the partitions. That is, p_2 gets the bit from p_1, p_3 gets the bit from p_2, etc. This is performed in two steps: odd partitions copy to even partitions, and then even partitions copy to odd [26] partitions, see Figure 6. We generalize the above single-shift to shifting all bits j partitions to the right (e.g, p_{j+1} receives from p_1, p_{j+2}, receives from p_2, ...) in $j+1$ steps with the ℓ^{th} step corresponding to all partitions p_i such that $i = \ell \mod (j+1)$. Notice that this is a deterministic shift (all rows shift by the same amount), unlike the variable shifting algorithm proposed in Section 4; Section 6 will combine these algorithms to attain bit-parallel variable shifting.

5.2.2 Broadcast

We desire to copy a single bit from a single partition (e.g., p_1) to all other partitions. This is achieved by first copying from p_1 to $p_{k/2+1}$, and then continuing recursively, in parallel, with the first half and the second half, as shown in Figure 6. The parallel execution is achieved by disconnecting the switch between $p_{k/2}$ and $p_{k/2+1}$. In total, $\log_2(k)$ steps [26].

5.2.3 Reduction

Each partition p_i stores a bit x_i, and we output $x_1 \circ \cdots \circ x_k$, where $\circ$ is any associative operation (e.g., AND, OR, XOR). This is achieved via a logarithmic tree, as illustrated in Figure 6. For example, in the first cycle, the partitions are split into adjacent pairs (e.g., p_1 and p_2 are connected, p_3 and p_4 are connected) and each section computes $\circ$ (storing the output in the partition with the larger index). The next cycle proceeds by connecting the partitions that contain the results in pairs (e.g., p_2 to p_4 are connected, p_6 to p_8 are connected). This continues following a logarithmic tree, until the last partition stores the overall reduction. Overall, $\log_2(k)$ steps in total.

5.2.4 Prefix

Each partition p_i stores a bit x_i as input, and each partition p_i outputs $y_i = x_1 \circ \cdots \circ x_i$, where $\circ$ is any associative operation (e.g., AND, OR, XOR). This generalizes the reduction technique as the last partition contains $x_1 \circ \cdots x_k$, and is far more complex as all partitions need to produce an output (the reduction of all bits up to that partition).

The naive algorithm will propagate through the partitions serially in $O(k)$ steps; that is, compute $y_2 = y_1 \circ x_2$ in p_2, then $y_3 = y_2 \circ x_3$ in p_3, and so forth. An improved

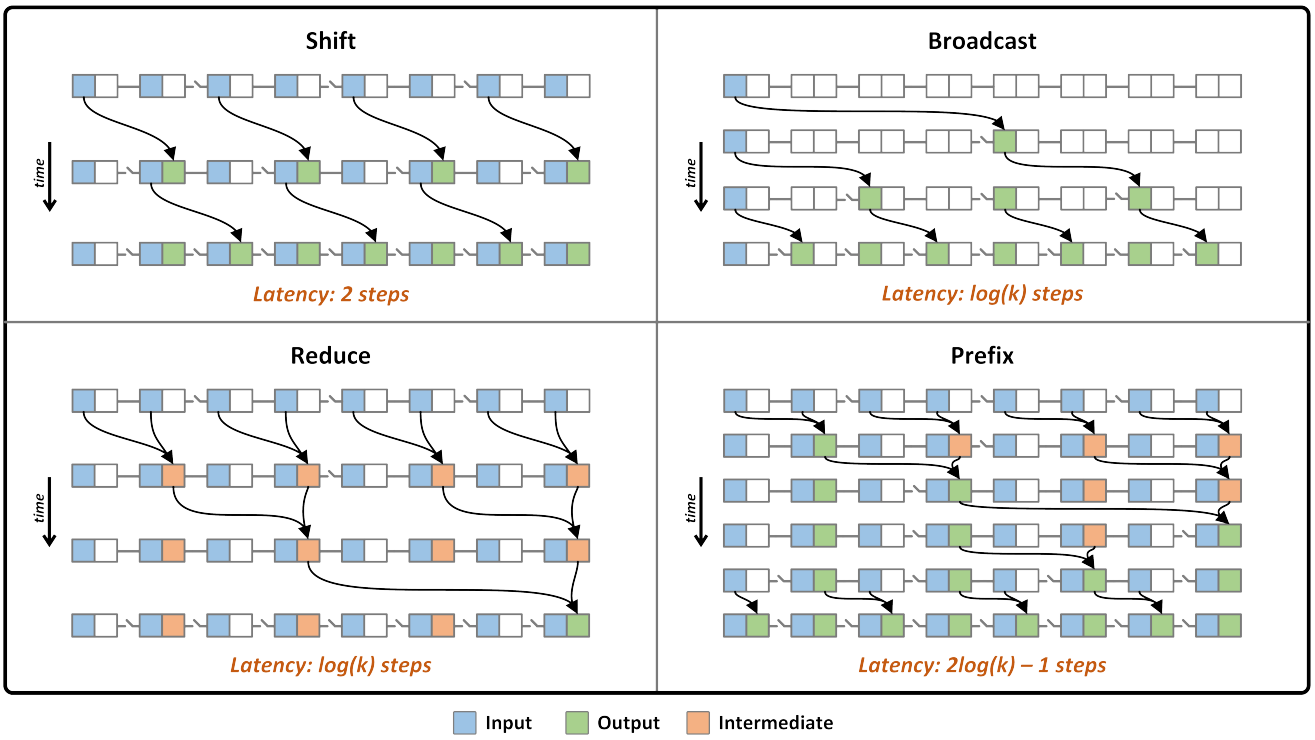

Fig. 6. The proposed partition toolbox, extending techniques proposed in MultPIM [26]. The illustrations follow a row of partitions (each shown as two cells for simplicity) as they progress throughout the cycles (vertical axis represents time); the states of the switches (connected or disconnected) are illustrated in the connections between the partitions (the connectivity at time t reflects the gates that occur between time t and time $t+1$). The shift technique shifts a single bit between neighboring partitions using two steps, the broadcast technique broadcasts a single-bit from one partition to all partitions using $\log_2(k)$ steps, the proposed reduction technique reduces (e.g., AND) bits from all partitions to a single partition using $\log_2(k)$ steps, and the proposed prefix technique computes for each partition the reduction of the bits from partitions before it using $2\log_2(k)-1$ steps.

algorithm relies on a recursive approach: first compute $y_{k/2}$ using the reduction technique in $O(\log_2(k/2))$ steps, broadcast that result to the upper $k/2$ partitions (as all of their prefix expressions include the reduction of the lower half of partitions), and then proceed recursively. Yet, the recursive approach require $O({\log_2}^2(k))$ steps total. Conversely, we propose an algorithm inspired by Brent-Kung [31] that only requires $O(2\log_2(k)-1)$ steps and is essentially based on the unique combination of a single reduction operation followed by a single broadcast operation. Intuitively, the intermediate results computed in the reduction resemble a prefix at some indices, and then the broadcast "fills in the holes" for the other indices by causing internal propagation. An example execution is shown in Figure 6, with the general algorithm provided in the code repository.

5.3 Bit-Parallel Fixed-Point Addition/Subtraction

This section demonstrates the first bit-parallel addition algorithm, inspired by parallel-prefix carry-lookahead adders [30], [31] and utilizing the prefix technique from the partition toolbox. The task is defined as follows: each row begins with two N-bit fixed-point numbers, x and y, stored in a strided format (i.e., each partition contains a single bit from x and a single bit from y), and the algorithm computes $z = x + y$ and stores it also in a strided format.

The parallel-prefix carry-lookahead adder was developed as a carry-lookahead design that utilizes the unique prefix operation for low latency with efficient area and energy; the reader is referred to [30] for a detailed explanation

Algorithm 5.1 Bit-Parallel Fixed-Point Addition

Input: N-bit x, N-bit y in a single row (strided format).
Output: N-bit result z in the same row (strided format), where $z = x + y$.

Pre-computation of alive and generate bits:
1: $\forall i \ : \ p_i.A \leftarrow \mathrm{OR}(p_i.x, p_i.y)$
2: $\forall i \ : \ p_i.G \leftarrow \mathrm{AND}(p_i.x, p_i.y)$
Prefix (using logarithmic prefix technique):
3: $\forall i \ : \ p_i.GG, p_i.AA \leftarrow (p_i.G, p_i.A) \circ \cdots \circ (p_0.G, p_0.A)$ where $(g,a) \circ (\tilde{g},\tilde{a}) = (g + a\tilde{g}, a\tilde{a})$.
Post-computation using shift and XOR:
4: $\forall i \ : \ p_{i+1}.c \leftarrow p_i.GG$
5: $\forall i \ : \ p_i.z \leftarrow \mathrm{XOR}(p_i.x, p_i.y, p_i.c)$

of the adder. Algorithm 5.1 attains this prefix derivation using the prefix technique[10] to propose the first in-memory bit-parallel addition algorithm. Overall, we require $O(\log(N))$ steps, improving the bit-serial state-of-the-art of $O(N)$ steps.

5.4 Bit-Parallel Fixed-Point Multiplication

This section introduces MultPIM [26], the state-of-the-art for bit-parallel multiplication which is based on the carry-save add-shift (CSAS) technique [43], and then improves the algorithm via the proposed bit-parallel addition algorithm.

The carry-save add-shift (CSAS) technique is a design for a latched multiplier circuit that computes the product of N-bit integers with N parallel full-adder units. The motivation for CSAS begins with carry-save addition: a technique that

10. Note that $\circ : \{0,1\}^2 \times \{0,1\}^2 \rightarrow \{0,1\}^2$ (each state is two bits). Therefore, the prefix technique is generalized to two bits per partition.

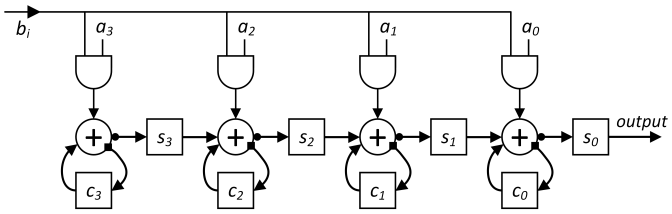

Fig. 7. The carry-save add-shift (CSAS) [26], [43] technique for multiplication. Circles are full-adders and squares are latches. Outputs of full-adders are marked as small circle (sum) and small square (carry).

Algorithm 5.2 Bit-Parallel Fixed-Point Multiplication

Input: N-bit x, N-bit y in a single row (strided format).
Output: N-bit results z, w in the same row (strided format), where $(w|z) = x * y$.

1: $\forall i \ : \ p_i.c, p_i.s \leftarrow 0$
2: **for** $i = 0, \ldots, N-1$ **do**
Broadcast of b_i to all partitions:
3: $\forall j \ : \ p_j.b' \leftarrow p_i.b$
Partial product computation:
4: $\forall j \ : \ p_j.ab \leftarrow p_j.a \cdot p_j.b'$
Carry-save addition, with shift:
5: $\forall j \ : \ p_j.s, p_j.c \leftarrow FA(p_j.s, p_j.c, p_j.ab)$
6: $\forall j \ : \ p_j.s \leftarrow p_{j+1}.s$
7: $p_i.z \leftarrow p_0.s$
8: **end for**
Proposed Final Addition using Alg. 5.1.
9: $w \leftarrow s + c$

avoids carry-propagation when computing the sum of many numbers. A carry-save adder receives three N-bit numbers, X, Y, Z, and outputs two N-bit numbers S, C, where $S + C = X + Y + Z$. This is done *without carry-propagation* using N full adders, where the i^{th} adder receives X_i, Y_i, Z_i and gives S_i and C_{i+1} (the i^{th} carry-bit becomes the $i + 1^{th}$ bit of C). To add many numbers, $X_1, \ldots, X_n$, we use carry-save adders to reduce it to a sum of only two numbers (fast as carry propagation is avoided), and then perform that using a single regular adder. Recall from Section 3.2 that multiplication can be expressed as the sum of many partial products; the fundamental idea of CSAS is using a carry-save adder for that addition. Similar to Section 3.2, the circuit exploits the zeros in the partial products and in the running sum. Figure 7 demonstrates the CSAS technique through a latched circuit design. For the i^{th} iteration out of N, the bit b_i is provided, all of the N full-adders perform the carry-save addition between the i^{th} partial product and the current running sum, and then all of the sum bits are shifted to the right and the last partition outputs $(a * b)_i$. These N iterations compute the lower N bits of $a * b$; for the upper N bits, either a regular adder is used, or an additional N stages feed zeros instead of b_i for N iterations.

MultPIM [26] utilizes the CSAS technique alongside the shift and broadcast techniques to propose an efficient bit-parallel multiplier. Each full-adder unit becomes a partition, thereby enabling the computation of all the full-adders in parallel. Furthermore, the partial products are also computed with parallelism as the broadcast technique is utilized to copy b_i to all of the partitions in the i^{th} iteration. Lastly, the shift technique is utilized to move the sum bits between the partitions efficiently. By performing N such iterations, MultPIM computes the lower N bits of the product. These steps are presented in Lines 1-8 of Algorithm 5.2.[11] To compute the upper N bits of the product, MultPIM proceeds with an additional N iterations where 0 is provided instead of b_i (full-adders are replaced with half-adders), instead of computing the sum of S and C directly. This choice was due to the best adder at the time requiring $O(N)$ cycles, and thus the latency was identical to performing an additional N iterations of $O(1)$ cycles each. Yet, the proposed bit-parallel adder from Section 5.3 can now be utilized instead, providing a significant advantage as $O(N)$ is reduced to $O(\log N)$ cycles and energy consumption (gate count) is reduced by approximately $1.6\times$. Algorithm 5.2 provides the proposed algorithm for bit-parallel multiplication which utilizes this optimization. Overall, we require $O(N \log(N) + \log(N))$ steps, improving over the $O(N \log(N) + N)$ of MultPIM.

5.5 Bit-Parallel Fixed-Point Division

We present the first bit-parallel divider by combining the concepts of *carry-save* and *carry-lookahead*. Division is far more complex than multiplication due to the conditional subtraction. While the conditional subtraction was avoided in Section 3.3 by replacing the control-flow with data-flow, the challenge here is that the sign of R (determined from the MSB) is not known if R is in carry-save format (represented as $R = S + C$). Therefore, this provides an inherent contradiction between the carry-save format and division. We overcome this by utilizing a carry-lookahead design similar to Section 5.3 to efficiently compute only the sign of $S + C$.

5.5.1 Carry-Save Carry-Lookahead (CSCL)

This section proposes the carry-save carry-lookahead (CSCL) technique for latched division which combines the carry-save technique from CSAS with carry-lookahead to predict the sign of R. The technique is based on a lesser-known design for parallel division arrays [32] that is traditionally not used due to the complex layout [30], [33].

The carry-save carry-lookahead (CSCL) division technique is shown in Figure 8, based on CSAS and Algorithm 3.3. A single-bit of z is inputted and a single-bit of q is outputted at each of the N iterations. Throughout the division, r is represented by s, c such that $r = s + c$ (carry-save format), and addition with r (Alg. 3.3, Line 3) is achieved via carry-save addition. The sign bit of r (Alg. 3.3, Line 4) is computed from s, c via carry-lookahead for the most-significant-bit. Shifting r (Alg. 3.3, Line 5) is achieved by shifting both s and c. The main aspects of CSCL are:

- *Carry-Save Format (Figure 8, black):* The remainder r is represented via s, c such that $r = s + c$, where s, c are stored in latches $s_1, \ldots, s_N$ and $c_2, \ldots, c_N$.
- *Carry-Save Addition (Figure 8, orange):* Similar to the bit-serial divider, the conditional addition/subtraction (Alg. 3.3, Line 3) is achieved via XOR (Alg. 3.4, Line 3). The addition is performed via the carry-save technique, utilizing N full-adders, and shifting the carry bits once (dashed orange arrows).

11. Note that MultPIM [26] was slightly modified to support strided format for both inputs and outputs.

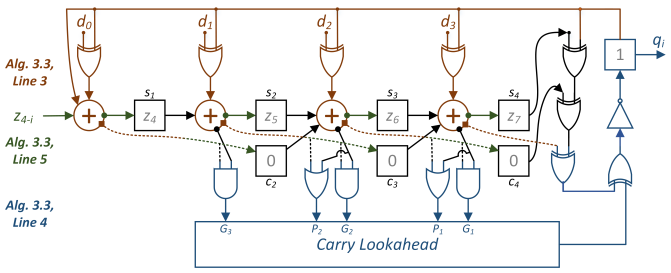

Fig. 8. The latched carry-save carry-lookahead division circuit for $N = 4$. Latches are squares and full-adders are circles. Solid/dashed line distinction indicates wires that do not intersect. Initial values are in gray.

Algorithm 5.3 Bit-Parallel Fixed-Point Division

Input: $2N$-bit dividend $(w|z)$, N-bit divisor d in a single row (strided format).
Output: N-bit quotient q, N-bit remainder r in the same row (strided format), where $(w|z) = qd + r$.

$\forall i \ : \ p_i.s \leftarrow p_i.w, p_i.c \leftarrow 0$
for $i = N-1, \ldots, 0$ **do**
  Broadcast of q_{i+1} to all partitions.
  $\forall j \ : \ p_j.q' \leftarrow p_{i+1}.q$
  Conditional addition/subtraction.
  $\forall j \ : \ p_j.dq \leftarrow \mathrm{XOR}(p_j.d, p_j.q')$
  $\forall j \ : \ p_j.s, p_j.c \leftarrow \mathrm{FA}(p_j.s, p_j.c, p_j.dq)$
  First carry shift.
  $\forall j \ : \ p_{j+1}.c \leftarrow p_j.c$
  Compute carry the carry of $s + c$:
  Same as Alg. 5.1 with prefix replaced with reduction.
  $p_i.q \leftarrow \mathrm{XNOR}(p_{N-1}.s, p_{N-1}.c, p_N.s, p_N.c, carry)$
  Remainder shifting:
  $\forall j \ : \ p_{j+1}.s \leftarrow p_j.s, p_{j+1}.c \leftarrow p_j.c$
end for
Final remainder computation using Alg. 5.1.
$r = s + c + \mathrm{AND}(d, \mathrm{NOT}(q_0))$

- *Carry Lookahead (Figure 8, blue):* The sign bit of r (required in Alg. 3.3, Line 4) is computed directly via carry-lookahead for the addition $s + c$.
- *Remainder Shifting (Figure 8, green):* The remainder shifting (required in Alg. 3.3, Line 5) is achieved by shifting both sum and carry bits once to the right.

5.5.2 Proposed Algorithm

The proposed CSCL technique is used in Algorithm 5.3. As in bit-parallel multiplication, each full-adder is represented via a partition, and inter-partition communication (e.g., carry-save, remainder shifting) is achieved with the toolbox (e.g., shifting, broadcasting). The carry-lookahead for the last carry is performed by modifying Section 5.3 to perform *reduction* rather than *prefix*. This computes the last carry in logarithmic time, and is faster than computing all carries as reduction is faster than prefix[12]. Overall, this is $O(N \log(N))$ steps, while the bit-serial state-of-the-art is $O(N^2)$.

6 Bit-Parallel Floating-Point Arithmetic

We merge the ideas from Section 4 (bit-serial floating-point) and Section 5 (bit-parallel fixed-point) for bit-parallel floating-point arithmetic. Recall that the algorithms from Section 4 relied on the variable shifting/normalization and fixed-point counterparts from Section 3. We now show *bit-parallel* variable shifting/normalization, and the extension to bit-parallel floating-point arithmetic follows by replacing the algorithms from Section 3 with those from Section 5. To allow fixed and floating point in the same crossbar, floating-point numbers are stored in strided format like fixed-point numbers (e.g., 32-bit floats are stored across 32 partitions).

Algorithm 6.1 Bit-Parallel Variable Shift Routine

Input: N_x-bit x, N_t-bit t, in a single row (strided format).
Output: N_x-bit $z = x \gg t$ in the same row (strided format).

$\forall i \ : \ p_i.z \leftarrow p_i.x$
for $j = 0, \ldots, \min(N_t - 1, \log_2(N_x) - 1)$ **do**
  Compute $z \leftarrow \mathrm{mux}_{t_j}(z \gg 2^j, z)$ as follows:
  Generalized shift technique:
  $\forall i \ : \ p_i.z' \leftarrow p_{i+2^j}.z$
  Broadcast technique:
  $\forall i \ : \ p_i.s \leftarrow p_j.t$
  Parallel multiplexer:
  $\forall i \ : \ p_i.z \leftarrow \mathrm{mux}_{p_i.s}(p_i.z', p_i.z)$
end for

We extend the bit-serial variable-shift and variable-normalization routines from Section 4.2 using the partition toolbox. For bit-parallel *variable-shift*, we utilize the generalized shift technique from the toolbox to get $z \gg 2^j$, and then use a parallel multiplexer (each partition performs a 2:1 1-bit multiplexer) to get $z \leftarrow \mathrm{mux}_{t_j}(z \gg 2^j, z)$. Algorithm 6.1 shows this in $O(\log(N_x) + \log^2(N_x) + N_x)$ steps (with a small constant in $O(\log^2(N_x) + N_x)$). For bit-parallel *variable normalization*, we also perform $t_j \leftarrow \neg(z_{N_x - 2^j} \vee \cdots \vee z_{N_x - 1})$ using the reduction technique from the partition toolbox. This provides bit-parallel variable normalization with the same complexity as bit-parallel variable shift.

7 Evaluation

We evaluate the AritPIM suite to verify the *correctness* of the algorithms, to *compare* its performance to previous PIM works and alternative solutions (e.g., GPU), and to facilitate its *adoption* by providing an open-access library with comprehensive implementations. The remainder of this section details the evaluation methodology and provides an overview of the results, while focusing primarily on the overall approaches. Further details on the experimental evaluation are available in the README of the code repository[13], alongside implementations for all of the algorithms.

7.1 Correctness

The correctness of the proposed algorithms is verified via a *cycle-accurate* simulation that consists of a PIM simulator and the library of the proposed algorithms. The PIM simulator models a single-row as a binary vector and possesses an interface for performing in-memory gates: a logic gate (e.g., NOR) may be sent to the simulator, and the simulator internally applies the logic gate. The library consists of implementations of the algorithms in this paper, each receiving the parameters of the algorithm (e.g., N) and outputting a sequence of in-memory gates. Together, correctness is verified as follows: the inputs are manually written to the PIM

12. This leads to the benefit of the proposed algorithm over merely implementing Algorithm 3.3 with Algorithm 5.1.

13. Available at https://github.com/oleitersdorf/AritPIM

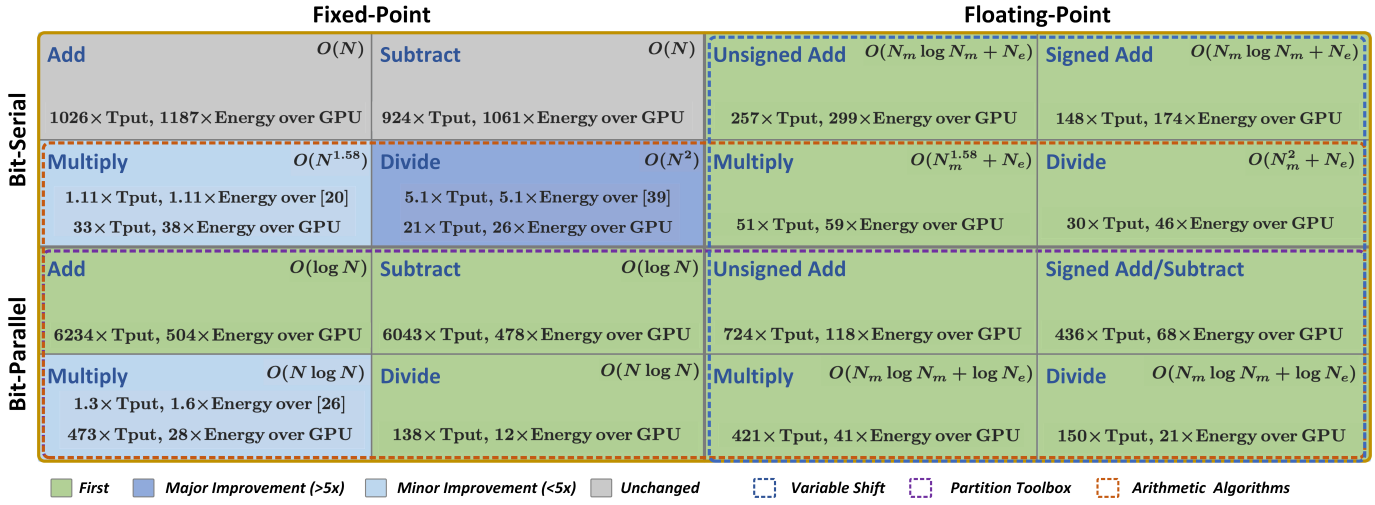

Fig. 9. Comparison of AritPIM to both the previous state-of-the-art for PIM (where relevant), and GPUs, for 32-bit numbers. The results compare both Throughput (arithmetic operations per second) and Throughput/Watt (energy). Additional results and details are provided in the repository.

simulator's internal memory (e.g., integers x, y), a sequence of logic gates is generated by the library and sent to the PIM simulator to be applied internally, and then the output (e.g., z) is manually read and compared to the expected value (e.g., $x + y$). For floating-point numbers, we compare to python-based floating-point operations (which adhere to the IEEE-754 standard). For partitions, we adopt the simulator for the minimal-model from PartitionPIM [27].

7.2 Performance Comparison

We compare the performance of AritPIM to previous PIM works (where relevant) and to GPU. Figure 9 summarizes the comparison, demonstrating both significant improvements over previous PIM works (in the few cases where previous works exist) and vast potential for high throughput compared to GPUs. Additional results for different parameters are available in the README of the code repository (e.g., cycle counts, energy, area); this section continues by detailing the methodology that derived these results.

For the comparison to alternative PIM works, we compare the cycle-count of the algorithms when implemented with the same underlying set of logic gates of NOT/NOR[14]. This provides a fair comparison between the algorithms as the underlying conditions are identical, and thus the only differences are the algorithmic concepts.

For the comparison to GPU, we consider a specific potential architecture of PIM and compare to experimental results from a modern GPU. Specifically, we consider a case-study of memristive PIM supporting the NOT/NOR gates [4], [5], with memristor parameters derived from RACER [5], constructing an 8GB memory from 1024×1024 crossbars.[15] The peripheral correctness and the evaluation of electrical limitations follow from ongoing works [4], [5], [44] that explore device and circuit models. These parameters reflect estimates of memristor performanc [5], and may differ between technologies; regardless, the overall trends and orders of magnitude remain and the proposed algorithms are directly applicable to all of the different forms of memristive PIM. Therefore, future work may select the most appropriate technology for PIM according to its specific parameters, and use the same algorithms from AritPIM. For GPU, we initialize vectors of $64M$ numbers in the GPU memory and measure the GPU performance when performing vectored operations (e.g., addition) on those vectors. Specifically, we utilize an NVIDIA RTX 3070 GPU with the `PyTorch` [45] profiling tools. Notice that this corresponds to data-intensive scenarios where the data does not fit within the cache; furthermore, we observed that the experimental results are almost identical to the theoretical upper-bound established by the GPU memory throughput – indicating that the memory wall is indeed the bottleneck. This observation also validates that other computational architectures (e.g., FPGA) subject to the same memory bandwidth will not outperform the observed GPU performance.

7.3 Adoption

One of the goals of this paper is to provide a foundational suite of arithmetic operations to advance PIM towards large-scale applications. Therefore, we have taken the following steps to facilitate the adoption of the proposed algorithms:

- *Uniformity:* All of the algorithms conform to the same interface for the inputs and outputs (e.g., $z \leftarrow x \circ y$ for $\circ \in \{+, -, *, \backslash\}$ all conform to the same interface). This simplifies the usage of arithmetic in a PIM architecture towards an application.
- *Open-Access Algorithms:* The implementations of the proposed algorithms are publicly available in the code repository. This enables their integration within cycle-accurate simulators for larger applications.
- *Verified Results:* We provide verified results for the cycle counts, energy, and area, of all of the algorithms, thereby enabling their usage in a larger application without requiring a cycle-accurate simulator.

14. Previous works were modified to assume 9 NORs per full adder to provide a fair comparison between the algorithmic concepts. Specifically, [20], [39] were upgraded from a 12-NOR full-adder to a 9-NOR full-adder, and the NOR-based implementation of [26] was adopted.

15. The evaluated vector dimension is $64M$ elements without loss of generality (as there are $64M$ rows in an 8GB memory of 1024×1024 arrays). Yet, AritPIM also supports larger vectors with identical Throughput and Throughput/Watt by using batches (e.g., $128M$-element vector addition is performed by storing two elements per row per vector, and then performing two $64M$-element vector additions serially).

8 Conclusion

As the memory-wall continues to limit the performance of modern computing systems, processing-in-memory (PIM) systems are rethinking the separation of storage and logic units to provide massive parallelism for bitwise logic within the memory itself. This paper extends this bitwise parallelism to high-throughput arithmetic in order to provide a foundation for large-scale PIM applications. We study the four elementary functions for both fixed-point and floating-point numbers, and via two emerging computational approaches of bit-serial and bit-parallel execution – providing the first algorithms in the literature for a majority of cases. Overall, this paper may be fundamental in the integration of large-scale applications with different PIM technologies.

Acknowledgments

This work was supported in part by the European Research Council through the European Union's Horizon 2020 Research and Innovation Programme under Grants 757259 and 101069336, and in part by the Israel Science Foundation under Grant 1514/17.

References

[1] 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, pp. 10–14.

[2] 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.

[3] 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.

[4] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (MAGIC)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.

[5] 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*, 2021, p. 100–116.

[6] 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, "SIMDRAM: A framework for bit-serial SIMD processing using DRAM," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, p. 329–345.

[7] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "ComputeDRAM: In-memory compute using off-the-shelf DRAMs," in *IEEE/ACM International Symposium on Microarchitecture*, 2019, p. 100–113.

[8] J. L. Potter, *Associative computing: a programming paradigm for massively parallel computers*. Springer Science & Business Media, 2012.

[9] [Online]. Available: https://www.gsitechnology.com/sites/default/files/Presentations/GSIT-Gemini-APU-Tech-Paper.pdf

[10] J. Wang, X. Wang, C. Eckert, A. Subramaniyan, R. Das, D. Blaauw, and D. Sylvester, "A 28-nm compute SRAM with bit-serial logic/arithmetic operations for programmable in-memory vector computing," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 1, pp. 76–86, 2020.

[11] 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, pp. 1–7.

[12] 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.

[13] 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*, vol. 17, no. 1, pp. 42–46, 2018.

[14] M. Zabihi, Z. Zhao, D. Mahendra, Z. I. Chowdhury, S. Resch, T. Peterson, U. R. Karpuzcu, J.-P. Wang, and S. S. Sapatnekar, "Using spin-hall MTJs to build an energy-efficient in-memory computation platform," in *20th International Symposium on Quality Electronic Design (ISQED)*, 2019, pp. 52–57.

[15] 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*, 2017, pp. 273–287.

[16] M. Imani, S. Gupta, and T. Rosing, "Ultra-efficient processing in-memory for data intensive applications," in *ACM/EDAC/IEEE Design Automation Conference*, 2017, pp. 1–6.

[17] V. Lakshmi, J. Reuben, and V. Pudi, "A novel in-memory wallace tree multiplier architecture using majority logic," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–11, 2021.

[18] R. Ben Hur, N. Wald, N. Talati, and S. Kvatinsky, "Simple magic: Synthesis and in-memory mapping of logic execution for memristor-aided logic," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 225–232.

[19] M. Zabihi, Z. I. Chowdhury, Z. Zhao, U. R. Karpuzcu, J.-P. Wang, and S. S. Sapatnekar, "In-memory processing on the spintronic CRAM: From hardware design to application mapping," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1159–1173, 2019.

[20] A. Haj-Ali, R. Ben-Hur, N. Wald, and S. Kvatinsky, "Efficient algorithms for in-memory fixed point multiplication using MAGIC," in *IEEE International Symposium on Circuits and Systems*, 2018.

[21] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky, "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*, vol. 39, no. 10, pp. 2434–2447, 2020.

[22] 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*, 2019, pp. 802–815.

[23] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, 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.

[24] J.-P. Wang and J. D. Harms, "General structure for computational random access memory (CRAM)," Dec. 29 2015, US Patent 9,224,447.

[25] Z. Lu, M. T. Arafin, and G. Qu, "RIME: A scalable and energy-efficient processing-in-memory architecture for floating-point operations," in *Asia and South Pacific Design Automation Conference*, 2021, pp. 120–125.

[26] O. Leitersdorf, R. Ronen, and S. Kvatinsky, "MultPIM: Fast stateful multiplication for processing-in-memory," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 3, pp. 1647–1651, 2022.

[27] O. Leitersdorf, R. Ronen, and S. Kvatinsky, "PartitionPIM: Practical memristive partitions for fast processing-in-memory," in *arXiv*, 2022.

[28] 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*, 2017, pp. 288–301.

[29] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, no. 2. Russian Academy of Sciences, 1962, pp. 293–294.

[30] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, 2010, vol. 2.

[31] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 260–264, 1982.

[32] M. Cappa and V. Hamacher, "An augmented iterative array for high-speed binary division," *IEEE Transactions on Computers*, vol. C-22, no. 2, pp. 172–175, 1973.

[33] M. Lu, *Arithmetic and logic in computer systems*, 1st ed., ser. Wiley Series in Microwave and Optical Engineering. Hoboken, NJ: Wiley-Interscience, 2004.

[34] Z. Sun, E. Ambrosi, A. Bricalli, and D. Ielmini, "Logic computing with stateful neural networks of resistive switches," *Advanced Materials*, 2018.

[35] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *IEEE International Symposium on High Performance Computer Architecture*, 2017, pp. 481–492.
[36] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *ACM/EDAC/IEEE Design Automation Conference*, 2016, pp. 1–6.
[37] H. Amrouch, D. Gao, X. S. Hu, A. Kazemi, A. F. Laguna, K. Ni, M. Niemier, M. M. Sharifi, S. Thomann, X. Yin, and C. Zhuo, "Ferroelectric FET technology and applications: From devices to systems," in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–8.
[38] D. Reis, M. Niemier, and X. S. Hu, "Computing in memory with FeFETs," in *International Symposium on Low Power Electronics and Design*, 2018.
[39] 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*, 2020, pp. 271–282.
[40] T. Jebelean, "Practical integer division with Karatsuba complexity," in *International Symposium on Symbolic and Algebraic Computation*, 1997, p. 339–341.
[41] 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*, vol. 33, no. 11, pp. 2872–2884, 2021.
[42] J. Li, R. K. Montoye, M. Ishii, and L. Chang, "1 Mb 0.41 µm² 2T-2R cell nonvolatile TCAM with two-bit encoding and clocked self-referenced sensing," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 4, pp. 896–907, 2014.
[43] R. Gnanasekaran, "A fast serial-parallel binary multiplier," *IEEE Transactions on Computers*, vol. C-34, no. 8, pp. 741–744, 1985.
[44] M. Zabihi, A. K. Sharma, M. G. Mankalale, Z. I. Chowdhury, Z. Zhao, S. Resch, U. R. Karpuzcu, J.-P. Wang, and S. S. Sapatnekar, "Analyzing the effects of interconnect parasitics in the STT CRAM in-memory computational platform," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 6, no. 1, pp. 71–79, 2020.
[45] 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.

Orian Leitersdorf (Student Member, IEEE) received the B.Sc. degree from the Technion, Haifa in 2022, and is currently a PhD candidate at the Technion, Haifa, Israel. He was a scholar at both the Technion Excellence Program and the Lapidim CS Excellence Program, he previously received the Gutwirth Excellence Scholarship, and he is currently a recipient of the Jacobs Excellence Scholarship. Further, he has received several awards, including the Early Career Researcher Paper Award at the ISITA 2022 conference. His current research aims to advance digital PIM towards fundamental applications (e.g., matrix operations, graph algorithms) while also addressing challenges such as reliability. Further, his research interests also include information theory and constrained coding.

Dean Leitersdorf received the B.Sc. and Ph.D. degrees from the Technion, Haifa, in 2019 and 2022. He was a recipient of several awards, including the 2018 Best Student Paper award at OPODIS, 2019 Best Student Paper award at PODC, the 2021 Jacobs Excellent Paper award (Technion), and the 2021 Jacobs Excellence certificate (Technion). His current research interests are in the fields of distributed graph algorithms, focusing on distance computations, sparse matrix multiplication and subgraph existence.

Jonathan Gal is currently studying towards his B.Sc in Computer Science and Mathematics at the Technion, Haifa, Israel, as part of the Technion Excellence Program. Jonathan Gal won a bronze medal in 2015 in the APIO (Asia Pacific Informatics Olympiad) and the IOI (International Informatics Olympiad). Between 2016 and 2020 he worked as a software engineer, focusing on image processing.

Mor Dahan is currently finishing his B.Sc. in Electrical Engineering at the Technion, Haifa, Israel. He joined Intel in 2018 as a DevOps engineer, and since 2020 he has been working as a hardware designer (focusing on pre-silicon verification).

Ronny Ronen (Fellow, IEEE) received the B.Sc. and M.Sc. degrees in computer science from the Technion, Haifa, Israel, in 1978 and 1979, respectively. He is a Senior Researcher with the Andrew and Erna Viterbi Faculty of Electrical & Computer Engineering at the Technion. He was with Intel Corporation from 1980 to 2017 in various technical and managerial positions. In his last role, he led the Intel Collaborative Research Institute for Computational Intelligence. He was the Director of microarchitecture research and a Senior Staff Computer Architect at the Intel Haifa Development Center until 2011. He led the development of several system software products and tools, including the Intel Pentium processor performance simulator and several compiler efforts. In these roles, he led/was involved in the initial definition and pathfinding of major leading-edge Intel processors. He holds over 80 issued patents and has published over 35 papers.

Shahar Kvatinsky (Senior Member, IEEE) is an Associate Professor at the Viterbi Faculty of Electrical and Computer Engineering, Technion. Shahar received the B.Sc. degree in Computer Engineering and Applied Physics and an MBA degree in 2009 and 2010, respectively, both from the Hebrew University of Jerusalem, and the Ph.D. degree in Electrical Engineering from the Technion in 2014. From 2006 to 2009, he worked as a circuit designer at Intel. From 2014 and 2015, he was a post-doctoral research fellow at Stanford University. Kvatinsky is a member of the Israel Young Academy. He is the head of the Architecture and Circuits Research Center at the Technion and chair of the IEEE Circuits and Systems in Israel. Kvatinsky has been the recipient of numerous awards including: 2020 MDPI Electronics Young Investigator Award, 2019 Wolf Foundation's Krill Prize, 2015 IEEE Guillemin-Cauer Best Paper Award, ERC starting grant, and the 2017 Pazy Memorial Award.