PimCity: A Compute in Memory Substrate featuring *both* Row and Column Parallel Computing

Salonik Resch*, Hüsrev Cılasun*, Masoud Zabihi, Zamshed Chowdhury,
Zhengyang Zhao, Jian-Ping Wang, Sachin S. Sapatnekar, Ulya Karpuzcu
University of Minnesota, Twin Cities
{resc0059, cilas001, zabih003, chowh005, zhaox526, jpwang, sachin, ukarpuzc}@umn.edu

***Abstract*—Processing-in-memory (PIM) substrates can perform parallel computation directly within the memory array. As a result, throughput performance and energy efficiency can reach unprecedented levels, however, there are limitations. Typical PIM architectures only support parallel computing in one dimension of the memory array: Computation is performed along either multiple rows or multiple columns (but not both). This restricts data layout and makes *intra-array* intermediate data transfers during computation inevitable - which require reads and writes, even for short-distance data movement. *Inter-array* data transfers, on the other hand, become a problem for larger scale algorithms which use more than one PIM array. Such data transfers incur large communication overheads and increase the complexity of the peripheral circuitry and interconnection network between arrays. Intermediate data transfers of any form limit scalability and efficiency. To overcome this limitation, we introduce *PimCity*, a new PIM substrate which can compute in *both* the rows and the columns of the memory array. PimCity replaces intra-array data transfer (memory) operations with lower overhead logic operations inside the memory array. Further, PimCity can perform logic operations directly across neighboring memory arrays, which in turn enables low-cost inter-array data transfers. PimCity hence represents a scalable PIM architecture suitable for both HPC and embedded IoT style applications.**

I. Introduction

The vast majority of modern applications from emerging domains such as machine learning or genomics have to process massive data quantities and hence are typically memory bound: the performance is limited by the rate at which data can be transferred between the processing logic and the supporting memory hierarchy. Therefore, according to Amdahl's Law no further increase in logic performance can cause an increase in the overall performance. Figure 1 showcases this problem, using matrix-vector multiplication, the cornerstone of machine learning inference and many other scientific applications, as a case study. Processing-in-Memory (PIM) was introduced in response to this problem. The idea is performing computation directly *within* the memory, such that data do not have to travel. In the ideal case, PIM enables *weak scaling*, where larger problems can be solved by simply adding more memory. Numerous PIM designs have demonstrated significant improvements for neural networks [12], [18], [30], pattern matching [7], graph processing [2], [9], [10], covering a wide range of applications [13], [21], [32], [37].

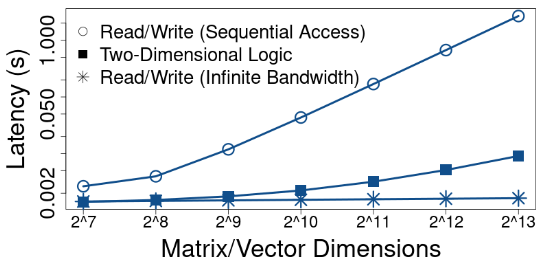


Fig. 1. Latency of in-memory matrix-vector multiplication with different data transfer capabilities. Communication becomes the limiting factor for performance. *Sequential access* for memory read/writes is the default in PIM systems supporting either row- or column-parallel computing but not both – which we refer to as 1D(imensional) parallelism. *Two Dimensional Logic* captures the proposed PimCity architecture enabling both row- and column-parallel computing, with 2D(imensional) parallelism. For reference, we also show the performance in the ideal case, under *Infinite Bandwidth* for read and write operations.

It is often overlooked that such PIM based hardware accelerators boil down to *distributed* systems with each node representing a PIM array. This is mainly because PIM array sizes cannot increase indefinitely (without hitting fundamental physical limits which can compromise reliability) while typical problem sizes tend to increase. As a result, there can be significant data movement throughout the memory structure as computation is in progress – not only within PIM arrays, but also between arrays. This easily can incur a large overhead. Fundamentally the requirement for data movement comes from data layout restrictions of PIM architectures regarding where inputs and outputs of a logic operation can reside in memory.

PIM architectures typically have either column parallelism [1], [21], [29], [32] or row parallelism [6], [42], as shown in Figure 2. With column (row) parallelism, logic operations are performed within each column (row) where inputs and outputs reside in separate rows (columns). Hence, data can be communicated (i.e., moved or transferred) between different rows (columns) but not between different columns (rows). As a result, depending on the computation, a large number of read and write operations may become necessary to move data to the appropriate locations in order to complete each logic operation. For larger scale algorithms distributed over more than one PIM array, such read and write operations are needed not only for moving data within the memory arrays (*intra-array*), but also between different memory arrays (*inter-array*), complicating the peripheral circuitry and interconnec-

*Authors contributed equally. This work was supported in part with Cisco fellowships.

tion network between arrays. *Be it intra- or inter- PIM array, intermediate data movement of any form limits scalability and efficiency.*

To overcome this fundamental limitation, we introduce a new PIM array architecture, *PimCity*, which can support parallelism in *both* dimensions (row and column). While PimCity cannot use row and column parallelism simultaneously, it can use one type of parallelism in one operation, and immediately switch to the other type of parallelism in the following operation. Having both types of parallelism enables arbitrary data communication (movement or transfer) within the memory array via *logic operations*, circumventing the need for data transfers via peripheral circuitry. For larger scale problems where multiple PIM arrays participate in computation, this renders a unique efficient tiled architecture. In this case, each tile consists of a single memory array and the minimal digital circuitry required to drive logic operations. Each tile has the ability to directly connect to its nearest neighbors in the cardinal directions. This allows logic operations to cross tiles, enabling inter-array data transfers to also be performed via logic operations. Most importantly, PimCity can handle such transfers in a highly parallel fashion, leading to high bandwidth communication without contending for a shared interconnection network. As a result, the complexity of PimCity scales well with number of tiles on chip, as each tile only needs to connect to its nearest neighbors.

Without loss of generality, in this paper, we focus on an especially promising class of PIM with unprecedented energy efficiency potential, which can perform universal (digital) Boolean computation directly in nonvolatile memory (NVM) [17], [19], [29]. This is in contrast to analog crossbar systems, which primarily perform weighted-sum operations. We should also note that there exist digital crossbar architectures [4], [33], [34] which support 2D (both column and row) logic operations, as well. However, a critical limitation of crossbar architectures, be it digital or analog, is the existence of sneak paths within the array [20]. Typically, the rows and columns of such crossbars are electrically connected together over the resistive memory devices in the array. Standard crossbars have no additional hardware dedicated to isolating rows or columns from each other. The large number of connections results in many, unintended (sneak) paths for current. The inability to completely isolate memory cells allows unwanted current to travel between different rows and columns during logic operations, which introduces noise and consumes additional energy. This noise can lead to incorrect operation and make the system unusable. To prevent excessive noise accumulation, digital crossbars typically have small dimensions and rely on RRAM devices, which are more resilient to noise due to their large resistance [33]. Lower endurance [16] and slower operation speed [24] of RRAM, however, when combined with small array sizes, limit practical use. Being restricted to small crossbars can make application mapping more challenging [4] and adds complexity to the peripheral circuitry at the same time. Many strategies have been proposed to mitigate sneak paths [20], such as including a single transistor [23], diode [43], or selector [39] within each memory cell. However, none of these strategies can eliminate the sneak path issue for both dimensions of computation.

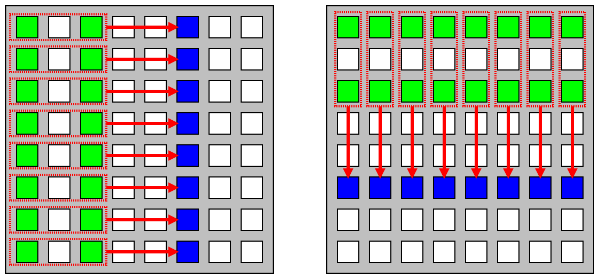


Fig. 2. Depiction of two-input logic gate operations, such as (N)AND or (N)OR, under (a) row and (b) column parallelism. Squares represent cells within the PIM array. Inputs are highlighted in green (light); outputs, in blue (dark). All rows in (a) and all columns in (b) perform the same computation on different data.

This paper is organized as follows: Section II covers the motivation; Section III, PIM basics; Sections IV and V, design specifics of PimCity; Section VI, a quantitative analysis underlining the critical nature of data movement in PIM architectures; Section VIII, related work; and Section IX, the conclusion and a discussion of our findings.

II. Motivation

A number of PIM architectures exist, which can perform Boolean logic in the memory array, including Ambit [32] for DRAM, X-SRAM [1] for SRAM, and Pinatubo [21] for NVM. All such substrates support one dimension of parallelism, either *row* parallelism or *column* parallelism[1], as shown in Figure 2. Architectures with row (column) parallelism can perform logic within each row (column) in parallel, with all inputs and outputs also residing in the same row (column). Effectively, each row (column) acts like a bit serial architecture, where each row (column) can compute in parallel with other rows (columns). Such architectures can be highly energy-efficient and capable of high performance. However, they have significant limitations. Unfortunately, in a row (column) parallel architecture, data cannot be transferred directly between rows (columns). Hence, data must be transferred with explicit reads and writes to enable further progress. Otherwise, each computation would be limited to data that can be stored in a single row (column).

For example, PIMBALL [30] is a column parallel digital PIM architecture which uses 1024×1024 memory arrays. This means each column is limited to using 1024 bits of data. As no realistically sized application requires only 1024 bits, many individual columns of the PIM architecture must work together – necessitating communication between them. This limitation extends to the array level, as well. A 1024×1024

[1]There is an important distinction between these digital PIM architectures and crossbars. Crossbars apply voltage along the rows and sense current in each column, and typically rely on analog computation [15], [26]. In contrast, digital PIM architectures perform digital computation, and each column is entirely isolated. In the case of PimCity, the logic is performed *in situ* where the inputs and outputs are cells in the memory array.

PIM array can hold 128KB of data. As this is insufficient for most purposes, inter-array data transfers are also required. As a consequence, PIM architectures use many read and write operations to move data *within* and *between* the memory arrays.

Such memory transfers can be costly, and conceptually break the motivating principle of PIM, hurting scalability. The transfers put heavy resource requirements on the peripheral circuitry and interconnection network. Worst of all, they can incur long distance data movement, even if there is no need for the data to travel far. For example, to shift data in a row by 1 bit (column) in a column-parallel architecture, the data must be read using sense amplifiers, transferred via interconnection network to external logic to perform the shift, transferred back via interconnection network after external logic is done, and finally written back to the original array. This is a large overhead for a simple operation. To complicate the matter further, if many such operations occur in many arrays simultaneously, they will compete for the interconnection network. *PimCity is capable of both types of computational parallelism shown in Figure 2, which, as we will demonstrate next, is critical in eliminating complications due to intermediate data transfers. Specifically, PimCity performs data transfers by simply using logic operations, as opposed to ordinary memory reads and writes.*

III. Background

We will next detail the design principles to enable 2D computational parallelism, which broadly apply to any 1D digital PIM performing Boolean logic directly in NVM such as MAGIC [19] or CRAM [41]. As a representative example, CRAM can support 1D column parallelism *without* using sense amplifiers [30]. This makes adaptation to 2D easier; we do not have to duplicate the sense amplifiers or re-route connections to them, as would be required for other types of 1D architectures [1], [21], [32].

CRAM uses Magnetic Tunnel Junctions (MTJ). MTJs are resistive memory devices that have two states, a low resistance state (logic 0) and a high resistance state (logic 1). The MTJ changes state when a sufficiently large current passes through it; the state it changes to depends on the direction of the current. MTJs provide higher density and higher endurance relative to other resistive memory technologies such as RRAM [40].

MTJs can perform logic efficiently when connected together in electrical circuits [28], as shown in Figure 3a. A (logic-gate) specific voltage is applied across the terminals V_1 and V_2, and the output MTJ will either switch or not depending on the states of the two input MTJs. For example, for a NAND gate the output MTJ is preset to 0. When voltage is applied and *both* of the input MTJs are 1 (high resistance), the high resistance of the inputs keeps the current low enough such that the output MTJ does *not* switch – it remains at 0. Otherwise, if *either* of the input MTJs is 0 (low resistance), the current becomes high enough to change the state of the output MTJ – it switches to 1. Hence, the state of the output MTJ reflects

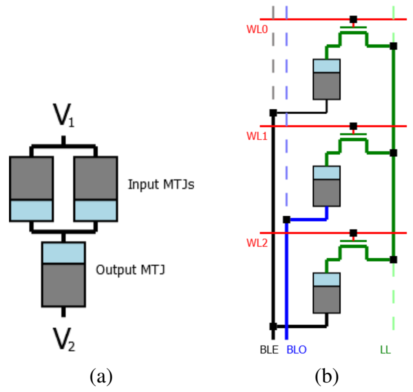


Fig. 3. a) Circuit for performing logic with MTJs. Two input MTJs in parallel are in series with an output MTJ. b) CRAM [30] can create equivalent logic circuits within the memory array. V_1 is applied to BLE (even bitline); and V_2, to BLO (odd bitline). LL (logic line) connects the input and output MTJs. WL indicates word lines.

the logical NAND of the states of the two input MTJs: 0 if both inputs are 1; and 1 otherwise. Other logic gates (NOT, AND, NOR, OR) can be performed similarly.

CRAM enables universal logic by generating the circuit in Figure 3a within the memory array [6], [30] as shown in Figure 3b. CRAM is effectively a lightly modified STT-MRAM array[2]. CRAM uses a single transistor per cell and contains three bitlines in each column, bitline even (BLE), bitline odd (BLO), and the logic line (LL). BLE (BLO) connects to MTJs in even (odd) parity rows. The LL connects to all MTJs through the access transistors. Activating multiple wordlines and applying a voltage across BLE and BLO can effectively perform logic in the memory, where the inputs and outputs are MTJs in different cells [30]. If limited to two input MTJs, such logic operations are robust [30], [42].

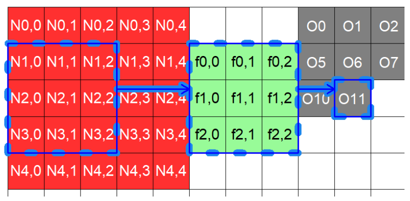
Sequences of such logic operations within the memory array can perform more complex operations [6]. Integer addition can be performed with sequences of full-adds on the individual bits, each of which can be completed using 9 NAND gates [30], [41]. Multiplication can be done with a dadda multiplier [36], which requires bit-wise AND gates followed by half-and full-adds [29]. Floating-point operations can likewise be mapped onto PIM substrates using such logic gates [8], although with less efficiency.

IV. Architecture

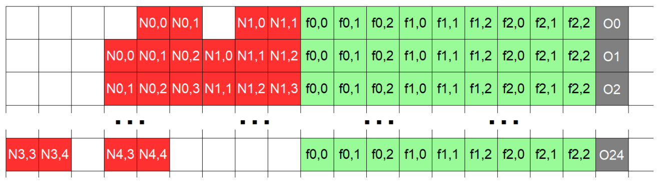
PimCity extends the 1D compute capability of CRAM to 2D, enabling both row and column parallelism. The hardware cost is one additional transistor per cell and two additional rowlines per row. Like CRAM, PimCity performs the logic *entirely* within the memory, not requiring the sense amplifiers or any external logic circuitry. This section covers the architecture of the memory array and details how each type of operation is performed within it.

Two-dimensional architectures can perform logic in both rows and columns, where the locations of the inputs and output are arbitrary. This enables them to perform all data

[2]CRAM could utilize RRAM cells as well, the only difference would be the energy efficiency of the operations and endurance of the cells.



(a) PimCity's 2D logic without data duplication


N0,0 N0,1
N1,0 N1,1
f0,0 f0,1 f0,2 f1,0 f1,1 f1,2 f2,0 f2,1 f2,2
O0
N0,0 N0,1 N0,2 N1,0 N1,1 N1,2
O1
N0,1 N0,2 N0,3 N1,1 N1,2 N1,3
O2
N3,3 N3,4
N4,3 N4,4
O24

(b) Necessary data duplication for 1D row logic [30]

Fig. 4. PimCity with 2D computing reduces data duplication requirements for application mapping. Computation (convolution) requires temporary values and multiple steps (not shown). N and f characterize neurons and filters; O, the result of computation. Using a combination of row and column logic, PimCity (b) operates on neurons and filters without duplication, requiring much less space than its 1D counterpart (a).

transfers directly, without any need for explicit read and write operations. *Effectively, data movement and logic operations are indistinguishable and use the exact same mechanisms, improving efficiency.* Additionally, as highlighted in Figure 4, this effectively eliminates any need for data duplication (which usually is required in 1D architectures for higher performance) as the data can be directly operated on in-place.

A. Memory Array

Each cell in the memory array contains a resistive memory element (MTJ), and two access transistors. Each column has 3 bitlines and each row has 3 wordlines. Two rows and two columns of a PimCity array are shown in Figure 5.

Each row has a Wordline for Column activation (WLC), Wordline for Row activation (WLR), and Row Logic Line (RLL). WLR controls an access transistor which connects each MTJ in the row to RLL. WLC controls an access transistor which connects each MTJ in the row to the Column Logic Line (CLL).

Each column has a Bitline Even (BLE), Bitline Odd (BLO), and Column Logic Line (CLL). BLE and BLO are connected directly to the MTJs, BLE in even parity rows and BLO in odd parity rows. CLL connects to all MTJs in each column, through the access transistor controlled by WLC. BLE and BLO serve an analogous role to the standard bitline in STT-MRAM, whereas the LL is analogous to the bitline bar.

Operations: Each memory cell must be capable of 5 different modes: (i) Retention, (ii) Read, (iii) Write, (iv) Column Logic, (v) Row Logic.

For *Retention*, no signals are applied. MTJs are non-volatile and maintain their state. PimCity performs *Read* and *Write* identical to standard STT-MRAM arrays. A single row can be read or written in an array at one time. To either read or write from row n, first WLCn is driven high. This connects MTJs in row n to the CLL in each column. Then, the MTJs can be accessed via the CLL and either BLE or BLO. If n

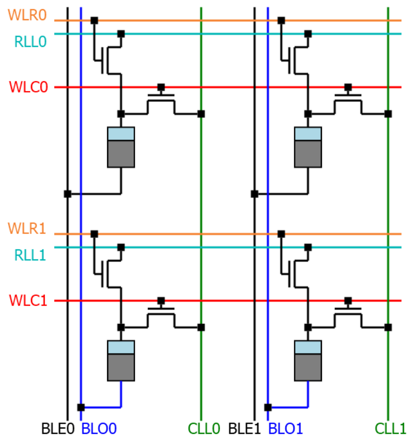


Fig. 5. Two rows and two columns of a PimCity array.

is even (odd), BLE (BLO) will be used. If the operation is a read, a low voltage is applied on BLE (BLO) and the resulting current can be sensed on the CLL. If the operation is a write, a sufficiently high voltage is applied on BLE (BLO), and the resulting current will overwrite the values stored in the MTJs.

To perform *Column Logic*, PimCity operates in an identical manner as CRAM [30], except that LL is now CLL; and WL, WLC (to differentiate them from RLL and WLR). For the logic operation, both inputs and the output are in the same column and on separate rows. Both inputs must be on the same parity rows and the output must be on the opposite (inputs on even rows and output on an odd row or inputs on odd rows and output on an even row). Let the two inputs be on rows a and b and the output be on row c. All we need is to drive WLCa, WLCb, and WLCc high, connecting the MTJs to CLL. Then, we need to apply a voltage differential across BLE and BLO. Current will travel from BLE(BLO), through the input MTJs, onto the CLL, through the output MTJ, and into BLO(BLE). Depending on the input MTJ states, the current will either be sufficient or not to change the output MTJ state. Hence, both CRAM and PimCity use BLE and BLO to supply the voltage. CRAM uses LL to connect the input and output MTJs, PimCity uses CLL. One operation can occur in each column at a time, but many columns can perform operations simultaneously, if they share the same input and output rows.

When performing *Row Logic*, the inputs and output are in the same row and in different columns. Let the inputs be in columns a and b and the output in column c. For a logic operation in row n, we first need to drive WLRn high to connect MTJs in row n to RLLn. Then we need to apply a voltage differential between BLE(BLO)a, BLE(BLO)b and BLE(BLO)c if n is even (odd). We can achieve this by setting both BLE(BLO)a and BLE(BLO)b to a gate specific voltage, while BLE(BLO)c is grounded. Current will travel from BLE(BLO)a,b through the input MTJs, onto RLLn, through the output MTJ, and out to BLE(BLO)c if n is even (odd). In this case, RLL connects the input and output MTJs (serving the same purpose as CLL for column logic or LL on CRAM). Again, only one operation can be performed in each row at a time. However, operations can be performed in many rows simultaneously, but the input and output columns must be the same. Logic can proceed in both even and odd parity rows simultaneously, as the same voltage can be applied to both BLE and BLO.

When performing logic, the line that connects MTJs together needs to be disconnected from the peripheral circuitry (connected to high impedance) to avoid current leakage. For column logic this applies to CLL; and for row logic, to RLL. Additionally, the BLE and BLO of columns that do not contain inputs or output are likewise disconnected.

Row and Column Activation: To perform computation in the memory, either in rows or in columns, both a set of rows and a set of columns have to be activated. One activation determines the input and the output MTJs, specifying on which rows (or columns) the inputs and the output reside. The other activation determines the degree of parallelism, specifying in which columns (or rows) the logic operation will occur. This directly applies to 1D PIM architectures, as well. For example, in a column-parallel architecture, the active rows determine the inputs and the outputs, and the active columns determine the degree of parallelism. PimCity as a 2D PIM substrate relies on the exact same semantics, except that the rows and columns can switch roles during computation according to algorithmic needs.

The standard method for activating a row or column is with a row or column decoder [21], [29]. A decoder takes an input address and then drives high the corresponding word or column line. We can activate multiple rows or columns without significant modification to the decoder, by adding a latching mechanism to each row or column and by supplying addresses to the decoder sequentially [21]. For example, if an operation has two inputs and one output, we can supply the three addresses sequentially. We use this approach for specifying the inputs and outputs. In order to activate many rows or columns simultaneously, we rely on bulk addressing [32], where a single (reserved) address corresponds to multiple rows or columns. This can be all rows or columns in an array. More reserved addresses can represent subsets of columns, for example one for the first half and another for the second half. Bulk addressing primarily serves setting the rows or columns to perform computation in. Frequently computations require activating all rows or columns in array. For computations in Figure 2b, as an example, we need to activate 3 rows (sequentially) and all columns (via bulk addressing).

B. Tiled Architecture

The PimCity array described in Section IV-A enables unique 2D compute capability. Here, we develop a tiled architecture in order to further exploit this functionality at scale, where large problem sizes necessitate distributing the computation across multiple PIM arrays. A single tile contains a PIM array and its corresponding row and column decoders. Each tile is effectively a self-contained processing unit. Applications of realistic sizes may easily require many tiles, which naturally incurs data transfers between tiles.

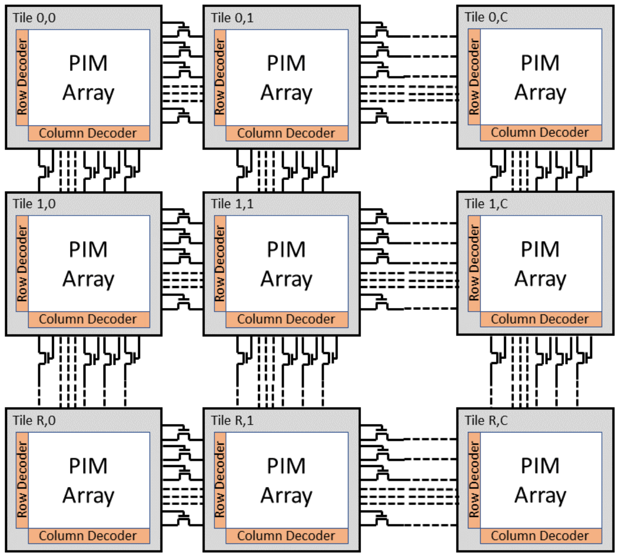


Fig. 6. The tile architecture of PimCity. The *column logic line* (CLL) and *row logic line* (RLL) of PIM arrays in neighboring tiles (in the cardinal directions) are connected via transistors. R is the number of rows of tiles and C is the number of columns of tiles.

Hardware Organization: The tile architecture expands the capability of performing intra-array data transfers via logic operations to inter-array data transfers. This is done by enabling neighboring tiles to connect their corresponding column logic lines and row logic lines together. The connection consists of transistors, one transistor for each row or column logic line, as shown in Figure 6. This connection enables a logic operation to have the inputs and the output in different PIM arrays. The 2D compute capability of the PIM array combined with the connection between tiles allows data to flow arbitrarily within and between tiles. However, the maximum distance between MTJs in a logic operation is limited by the parasitic resistance and capacitance of the bitlines [42]. Hence, inter-tile logic operations are limited to neighboring tiles.

As the tiles only need to be connected to their nearest neighbors, the hardware cost of the connections increases linearly with the number of tiles. Hence, this architecture can easily scale to accommodate many such tiles, and can especially be beneficial for applications featuring weak scaling. We consider the case where all tiles are integrated on chip.

Instructions and Control: For the tiles to work together (by transferring data between each other via logic operations), it is a strict requirement that they operate synchronously. Hence, all tiles are driven by a single controller. In this work, we use a configuration similar to that in MOUSE [29], which also has a tile-based PIM architecture, but 1D. In MOUSE, a controller broadcasts customized PIM instructions to the tiles, each specifying the logic operation (such as NOT, NAND, NOR) and the row addresses to perform them on. We use a nearly identical instruction set, except that we require an additional bit specifying whether the operation should be row or column parallel.

Trading Latency for Area Efficiency: Regarding system integration, thus far, we highlighted PimCity as a PIM architecture, where, despite significant computational capability, memory functionality remains intact. We can also design PimCity more aggressively to operate purely as an accelerator by removing read functionality from the majority of the tiles. As write and logic operations do not require sense amplifiers, they can be removed, along with any additional peripheral circuitry required for reads. This saves area. Then, in order to extract data from the tiles that do not have sense amplifiers, we can use inter-tile logic operations to transfer the data to a subset of tiles which do contain sense amplifiers. Once located in tiles that have sense amplifiers, we can read out the data.

V. Putting It All Together

We next provide two case studies to demonstrate the effectiveness of PimCity in reducing intra- and inter-array data transfers, respectively. Without loss of generality, we use a row parallel architecture (logic operations are performed within individual rows of the memory array) as a baseline for comparison.

Case Study I: We first illustrate how PimCity can reduce intra-array communication overhead relative to 1D architectures. Assume that we want to add two 4-bit integers, $Y = y_3y_2y_1y_0$ and $X = x_3x_2x_1x_0$. The data layout is provided in Figure 7. In this case, in a row parallel (1D) PIM architecture we need to read out either X or Y and then write it back to the same row as the other operand, before we can perform the actual computation. PimCity, on the other hand, featuring both row and column parallelism, offers two alternatives: The first is to simply perform the addition without any data movement, with a combination of row and column logic operations. The second, as depicted in Figure 7, is to transfer X to the same row as Y via logic operations: We can use NOT gates to generate X' in a spare set of columns. Then, a second round of parallel NOT gates can place X into the same row as Y. Computation can then proceed with exclusively row logic operations, as in a row parallel architecture. This logic-based method of transferring data takes $b + 1$ steps, where b is the bit-precision of the numbers (4 in the example). Such intra-array data transfers via logic operations in PimCity are highly energy efficient as the underlying logic gates are highly energy efficient and there is no need for the data to leave the array. Significantly, we can trigger intra-array data transfers in each array simultaneously.

Case Study II: We next cover matrix-vector multiplication as a critical computational kernel for machine learning inference and many other scientific applications, using two PimCity tiles as shown in Figure 8. Without loss of generality, we assume that the matrix (M) has 7 rows and 2 columns and already resides in the memory. The matrix is to be multiplied with a 2-element vector (V). First, we must write the elements of V into the tiles as in Figure 8a. Each element of V is to be multiplied with each column of M, hence we need to duplicate the elements of V, one copy for each row of M. As PimCity supports column logic, we can copy the elements to each row with logic operations – without using additional write operations, as shown in Figure 8b. Multiplication of the matrix and vector elements can then proceed in each row simultaneously using row logic operations, as depicted by Figure 8c. Each multiplication requires a sequence of bitwise ANDs and full-adds, all of which can be implemented with simple NAND operations. Multiplications produce partial sums (P), which need to be added together. Row-logic operations, as shown in Figure 8d, can copy partial sums to the same tile. Once in the same tile, partial sums can be added together to form the final sum (S), as in Figure 8e. Addition also requires full-adds, which can be implemented with NAND gates (Section III). Computation is then complete and the result is ready to be read out.

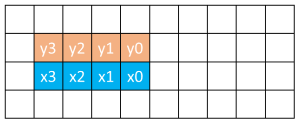
VI. Evaluation Setup

Benchmarks: We evaluate PimCity on large scale matrix-vector multiplication. This is an insightful benchmark, as matrix-vector multiplications are at the core of the vast majority of modern workloads and scientific applications, including neural networks [22], [38], support vector machines [5], [29], and quantum simulation [35]. For example, in MOUSE, over 90% of the latency and over 99% of the energy of support vector machine inference in PIM goes to matrix-vector multiplication [29]. As another example, for YodaNN [3], a hardware accelerator for neural networks, over 70% of the energy goes to the weighted multiplication of neurons and weights, a computation highly similar to matrix-vector multiplication.

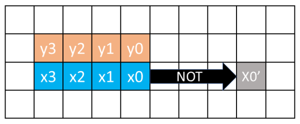
We perform multiplication of a matrix with 10 different vectors. We use 32-bit precision fixed-point numbers and test with matrix sizes from 256×256 up to 8,192×8,192. At the start, the matrix must be written into the memory, from then on it can be reused for further computations. This is representative of most applications, where the matrix would typically be stored in the memory prior and kept constant. This is the case for weights for neural networks, and the support vectors for support vector machines [30]. The vectors must be written in and read out for each matrix-vector multiplication. We use 10 sequential matrix-vector multiplications (and consequently 10 vector write-ins and read-outs) to capture the overhead associated with data movement required for input and output.

Technology Parameters, Modeling, Simulation: Our estimates for MTJ write energy and latency come from [27], [31], which provide the latency and required current for MTJ writes from experimental demonstrations, as summarized in Table I. Please refer to [25] for a detailed discussion on projected parameters. To model peripheral circuitry overhead, we take estimates of the relative latency and energy of memory array access from NVSIM [11]. As we require multiple row or column activation, we scale up the cost of the peripheral circuitry accordingly as described in Section IV-A. If a logic operation requires the activation of 3 rows (e.g., for a NAND gate), we assume that the peripheral latency and energy costs are both 3× the cost of a single access.

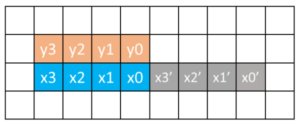
Baselines for Comparison: We compare PimCity with two PIM architectures:



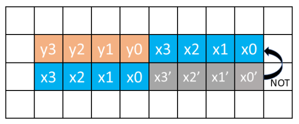
(a) Initial layout



(b) Row-wise NOTs to generate X' in the same row as X

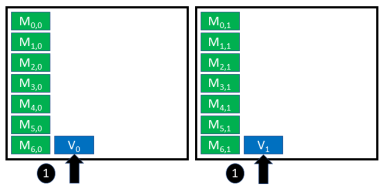


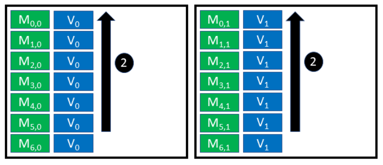
(c) X' obtained after 4 NOTs

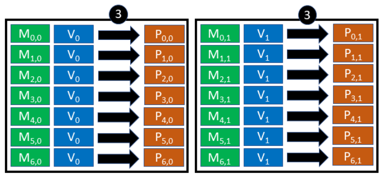


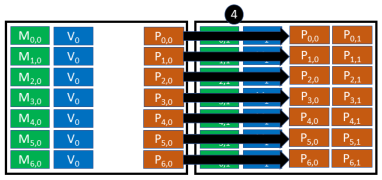
(d) Column-wise NOTs to copy X to the same row as Y

Fig. 7. Case Study I: Moving two 4-bit integers, X and Y, onto the same row via logic operations. Once on the same row, row logic operations can be used to work with both X and Y.


(a) Write In


(b) Duplication via column logic


(c) Multiplication with row logic


(d) Copy with inter-tile row logic

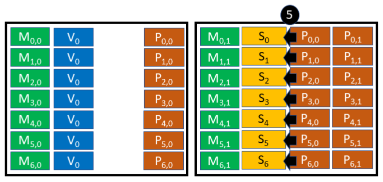

(e) Sum with row logic

Fig. 8. Case Study II: Matrix-vector multiplication of a 7×2 Matrix (M) and 2-element vector (V) using two PimCity tiles. Multiplications, additions, and copies are performed with sequences of simple logic operations in memory [6], [30], [41], as described in Section III. (a) Vector elements are written into a single row. (b) Column logic is used to copy the vector elements to all rows. (c) Multiplication occurs in each tile simultaneously with row logic operations, generating partial sums (P). (d) Inter-tile row logic gates copy partial sums to same tile. (e) Row logic is used to sum partial sums into final sum (S).

TABLE I
MTJ PARAMETERS

Parameter	Value
R_P	7 kΩ [14]
R_{AP}	70 kΩ [14]
I_{switch}	100 µA [27]
t_{switch}	3 ns [31]

i. A highly optimized, 1D CRAM-based, representative and practical design from [29], which we will refer to as *Entirely Sequential* or *Seq* for short;
ii. A hypothetical 1D design reflecting the theoretical best case for 1D PIM, which we will refer to as *Entirely Parallel* or *Par* for short.

We use identical latency and energy estimates for MTJ operations and peripheral circuitry accesses for all designs, hence there are no differences due to technology parameters. Differences in performance come strictly from the architecture.

Entirely Sequential or *Seq* from [29] was designed for small, low-power applications and hence it has a modest interconnection network. It is an MTJ (CRAM) based PIM substrate which allows reads and writes to all memory arrays, but only one at a time. It has a 1D PIM substrate, which uses column-parallel logic (in contrast to PimCity which has both column-parallel and row-parallel logic). As only one tile can be accessed at a time, the reads and writes to transfer data must be performed entirely sequentially. Hence, we refer to this configuration as *entirely sequential*.

Entirely Parallel or *Par* is an ideal 1D PIM architecture which uses reads and writes for data transfers. We take a 1D row-parallel PIM architecture and assume that it can perform reads and writes to *all* arrays simultaneously, without any additional latency or energy cost. If there are X PIM tiles, a row from each of the X tiles can be read in one cycle, and then written to any other row of any of the X tiles in the next. We also assume that the data can be arbitrarily permuted between the reads and writes without any additional overhead. This allows for arbitrary data movement (of up to one row in each tile) in two cycles. Effectively, this represents the theoretical best case for a 1D PIM architecture with an unlimited I/O bandwidth (it does pay the same latency and energy cost per memory access, but we do not impose any additional penalty for the unlimited bandwidth or data rearrangement); hence the name, *entirely parallel*.

In any given system, the maximum number of simultaneous reads and writes depends on the interconnection network and how effective latency masking techniques such as buffering can get. The two configurations we consider, *entirely sequential* and *entirely parallel*, represent the extreme ends of this design space. The performance of any practical PIM architecture (which uses standard reads and writes for data transfer) should fall between the two.

Hardware Configuration (Data Layout & Sizing): We test with two tile sizes, 256×256 and 1024×1024. Due to circuit-level restraints (as induced by bitline resistance and capacitance), a reasonable assumption for the maximum size for an MTJ PIM array is 1024×1024 [42]. For other MTJ (specifically, variants of CRAM [42]) based PIM architectures which use two transistors per cell, the cell area can reach $0.044\,\mu\mathrm{m}^2$ [42]. The area overhead is based almost entirely on the number of transistors per cell, as they are much larger than the MTJs [42]. Using this cell area estimate, a 1024×1024 PIM array in PimCity would consume $0.046\,\mathrm{mm}^2$. The additional area overhead for the connections between tiles is negligible, as each connection requires only 1024 transistors (for a 1024×1024 array, which already requires 2,097,152 access transistors). Table II lists the area overhead for different tile and problem sizes.

All of the architectures use the same data layout from [30]. The authors of [30] mapped neural networks (consisting of mostly matrix-vector multiplication) to a CRAM-based 1D PIM accelerator, using standard read and write operations for data transfer. Hence, the data layout is optimized for 1D PIM architectures using standard reads and writes. This gives an optimal configuration for the *entirely parallel* and *entirely sequential* designs. The tile-based 2D PimCity does not directly benefit from this customized data layout.

For each configuration, we use the minimum number of tiles required to fit the matrix (**M**), a single vector (**V**), and sufficient extra space for temporary values. The elements of the matrix are placed row-wise into the rows of the tiles. Assuming PIM tile dimensions are $T \times T$, for an $n \times n$ matrix, a total of n rows are needed, which requires $\left\lceil \frac{n}{T} \right\rceil$ rows of tiles. Each row of each tile needs sufficient space for the matrix and vector elements, plus some extra space to hold temporary values during computation. The number of elements that can be placed in each row, t, depends on the tile size, T.

Figure 9 captures the data layout. The process is highly similar to Case Study II from Figure 8. The matrix and vector elements are placed into the array, using writes (and column logic for PimCity). The matrix and vector elements are element-wise multiplied within each row with row-parallel logic operations. After multiplication, the elements within each tile row are added together to form partial sums. Once addition is complete within each row, inter-tile logic operations (for PimCity) or reads and writes (for *entirely sequential* and *entirely parallel*) transfer the partial sums between tiles to be combined further. After $log(n)$ additions, only a single sum remains within each row, which is the output of the computation.

VII. Results

Figure 10 shows the latency for matrix-vector multiplication for different input dimensions and different tile sizes. As expected, the latency grows rapidly for the *entirely sequential* configuration, *Seq*, due to drastically increasing communication overhead with problem size. On the other hand, overall, PimCity's performance comes very close to that of the *entirely parallel* configuration, *Par*. Most importantly, with larger tile sizes and smaller input matrices, PimCity can actually beat the *entirely parallel* configuration *Par*. This is due to the parallelism of the data transferring logic operations in PimCity. While *Par* can read or write from every tile simultaneously, it must access the rows sequentially. If a tile contains T rows, it must perform T reads to access all data. If the tile is relatively large, the large number of sequential reads and writes result in considerable latency, regardless of the speed of the interconnection network. In contrast, as PimCity can transfer data with row-parallel logic operations, the data in each row can be transferred to a neighboring tile in parallel. However, in this case, the bits in each row must be transferred sequentially (in contrast to *Par* which can read all bits in a row at once).

As the problem size gets larger, *Par* gains a significant advantage. As the tile size cannot increase to match the problem (due to circuit limitations), larger problems require many more tiles. When there are many tiles, the limitation becomes the bandwidth of long distance data transfers. PimCity performs well, delivering good performance for very large problems. However, long distance data movement involves many tiles, hopping between neighboring tiles (again due to circuit limitations). Hence, as the number of tiles increases significantly, the latency of PimCity also increases. In contrast, *Par* can move data in two steps, a read from the source tile and a write to the destination tile. As the tile sizes cannot grow indefinitely, the number of sequential reads and writes to access data from a single tile does not increase with problem size. Therefore, *Par* has a very minor increase in latency with increasing problem size. We should also note that, due to the idealized assumptions, *Par* represents *weak scaling*, where hardware resources proportionally grow with the problem size. This analysis accounts for the overhead of memory controllers.

We next quantify the relative share of *computation* and *communication*. Computation refers to all operations performed in memory to implement multiplications and additions, to guarantee forward progress. Communication represents all data movement operations required to enable computation. This includes logic operations that transfer data between arrays in PimCity and the intermediate reads and writes that are used to transfer data between arrays in the *entirely sequential* (*Seq*) and *entirely parallel* (*Par*) designs. For all designs, communication also includes writes to store the input in the memory and reads required to extract the results from memory.

The breakdown of latency for computation and communication for a practical tile size of 1024×1024 with an 1024×1024 input matrix is shown in Figure 11a. As all designs need to perform the same computations, the latency for computation is nearly identical. However, the latency for communication varies considerably due to the differences in the interconnection network. Notably, *Seq* has a very large latency overhead, and PimCity is able to achieve a similar latency to *Par*.

The corresponding breakdown of the energy consumption is shown in Figure 11b. For all designs and array sizes, computation dominates the energy consumption. This is because many computations take place between each data transfer. Computation is also highly-parallel, and thus consumes relatively higher energy. If communication is highly-parallel as well, as it is the case for *Par* and PimCity, it also has low latency and

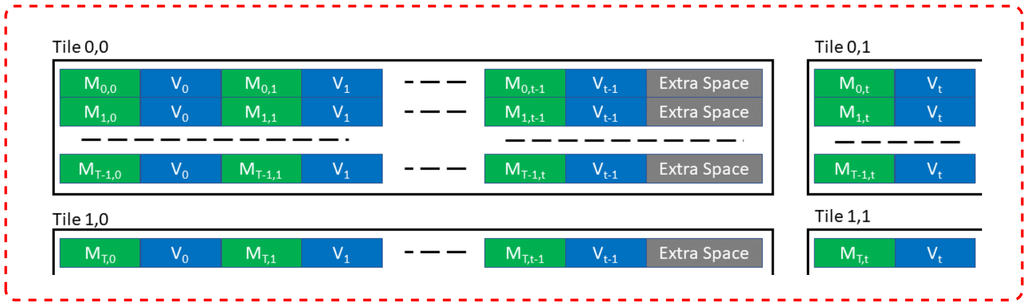

Tile 0,0
$M_{0,0}$ V_0 $M_{0,1}$ V_1 $M_{0,t-1}$ V_{t-1} Extra Space
$M_{1,0}$ V_0 $M_{1,1}$ V_1 $M_{1,t-1}$ V_{t-1} Extra Space
$M_{T-1,0}$ V_0 $M_{T-1,1}$ V_1 $M_{T-1,t}$ V_{t-1} Extra Space
Tile 0,1
$M_{0,t}$ V_t
$M_{1,t}$ V_t
$M_{T-1,t}$ V_t
Tile 1,0
$M_{T,0}$ V_0 $M_{T,1}$ V_1 $M_{T,t-1}$ V_{t-1} Extra Space
Tile 1,1
$M_{T,t}$ V_t

Fig. 9. The elements of a matrix (M) and a vector (V) laid out in memory. There are T rows in each tile. The number of elements that can be fit in each row, t, is a function of the tile size. Many tiles may be required to hold all data. $M_{i,j}$ is the matrix element in the ith row and jth column. V_i is the ith element of the vector.

TABLE II
THE NUMBER OF TILES AND THE CORRESPONDING AREA IN MM^2 FOR DIFFERENT INPUT MATRIX SIZES WHEN USING 256X256 AND 1024X1024 TILES.

Tile Size vs. Matrix Size	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
256	43 / 0.12	86 / 0.25	342 / 0.99	1368 / 3.94	5464 / 15.76	21856 / 63.02	87392 / 252.00
1024	9 / 0.42	18 / 0.83	35 / 1.61	69 / 3.18	274 / 12.64	1096 / 50.57	4376 / 201.90

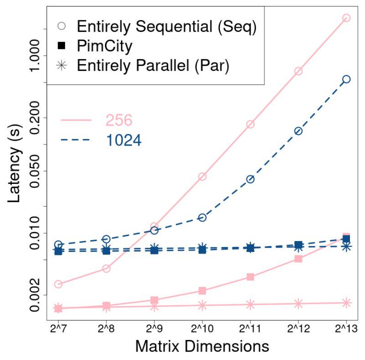
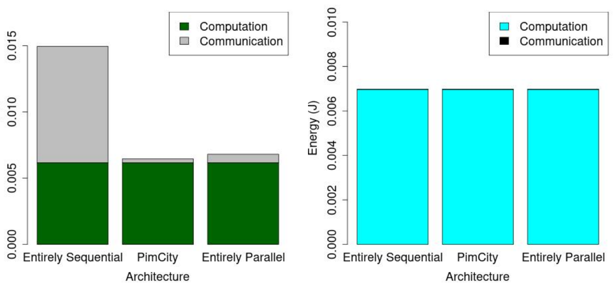


Fig. 10. Latency of 10 matrix-vector multiplications for different matrix dimensions. Shown are two different tile sizes, 256x256 and 1024x1024.



(a) Latency (b) Energy

Fig. 11. Latency and Energy breakdown for a practical tile size of 1024×1024 with input matrix dimension of 1024×1024. Communication accounts for a larger percentage of the latency but less than 1% of the energy.

hence consumes relatively low energy (but high power). If communication is highly sequential, as in *Seq*, data transfers consume very low power but have a high latency.

VIII. Related Work

Very energy efficient PIM based accelerators for matrix multiplication [10], [26] and dot-products [15] exist. However, these designs rely on analog computations with many inputs, making them highly susceptible to noise. As a consequence, such designs can only map small and low bit-precision computations onto the PIM hardware. Much of the computation must be performed on external, digital circuitry. PimCity has digital PIM operations, and hence can scale to significantly larger problem sizes or bit precisions. Additionally, all computation is performed in the memory.

IX. Conclusion

Due to data layout and array size limitations of PIM substrates, ever growing data sizes of emerging applications render intra- and inter-PIM array data transfers inevitable. Unfortunately, such data transfers impair scalability. In this work we presented PimCity, a PIM substrate which can perform Boolean logic in both the rows and columns of its memory arrays. Using this capability, PimCity can perform intra- and inter-array data transfers by simple logic operations. This is in stark contrast to ordinary PIM designs (which can compute only in 1D, i.e., in either rows or columns) which have to rely on explicit memory read and write operations for any type of data transfer. As a result, since in-memory logic operations are much more energy efficient than ordinary memory read and writes, PimCity can cut the overhead of data transfers significantly.

References

[1] A. Agrawal, A. Jaiswal, C. Lee, and K. Roy, "X-sram: Enabling in-memory boolean computations in cmos static random access memories," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2018.
[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *42nd Annual International Symposium on Computer Architecture*, 2015.
[3] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An ultra-low power convolutional neural network accelerator based on binary weights," in *IEEE Symposium on VLSI (ISVLSI)*. IEEE, 2016.
[4] D. Bhattacharjee *et al.*, "Contra: area-constrained technology mapping framework for memristive memory processing unit," in *ICCAD*, 2020.
[5] C.-C. Chang *et al.*, "Libsvm: a library for support vector machines," *ACM transactions on intelligent systems and technology*, 2011.

[6] 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, 2017.

[7] Z. I. Chowdhury, S. K. Khatamifard, Z. Zhao, M. Zabihi, S. Resch, M. Razaviyayn, J.-P. Wang, S. Sapatnekar, and U. R. Karpuzcu, "Spintronic in-memory pattern matching," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 5, no. 2, 2019.

[8] H. Cılasun, S. Resch, Z. I. Chowdhury, E. Olson, M. Zabihi, Z. Zhao, T. Peterson, J.-P. Wang, S. S. Sapatnekar, and U. Karpuzcu, "Crafft: High resolution fft accelerator in spintronic computational ram," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020.

[9] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "Graphh: A processing-in-memory architecture for large-scale graph processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 640–653, 2018.

[10] G. Dai, T. Huang, Y. Wang, H. Yang, and J. Wawrzynek, "Graphsar: a sparsity-aware processing-in-memory architecture for large-scale graph processing on rerams," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, pp. 120–126.

[11] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.

[12] 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 *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.

[13] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 1–14, 2018.

[14] A. Hirohata *et al.*, "Roadmap for emerging materials for spintronic device applications," *IEEE Transactions on Magnetics*, 2015.

[15] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams, "Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[16] X.-D. Huang *et al.*, "Forming-free, fast, uniform, and high endurance resistive switching from cryogenic to high temperatures in w/alo x/al 2 o 3/pt bilayer memristor," *IEEE Electron Device Letters*, 2020.

[17] M. Imani, Y. Kim, and T. Rosing, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017.

[18] L. Jiang, M. Kim, W. Wen, and D. Wang, "Xnor-pop: A processing-in-memory architecture for binary convolutional neural networks in wide-io2 drams," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2017, pp. 1–6.

[19] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2014.

[20] H. Li *et al.*, "Memristive crossbar arrays for storage and computing applications," *Advanced Intelligent Systems*, 2021.

[21] 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 *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[22] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "Fp-bnn: Binarized neural network on fpga," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018.

[23] H. Liu *et al.*, "Uniformity improvement in 1t1r rram with gate voltage ramp programming," *IEEE Electron Device Letters*, 2014.

[24] J. Louis *et al.*, "Performing memristor-aided logic (magic) using stt-mram," in *ICECS*, 2019.

[25] Y. Lv and J.-P. Wang, "Manufacturability evaluation of magnetic tunnel junction-based computational random access memory," *IEEE Transactions on Magnetics*, 2023.

[26] L. Ni, Y. Wang, H. Yu, W. Yang, C. Weng, and J. Zhao, "An energy-efficient matrix multiplication accelerator by distributed in-memory computing on binary rram crossbar," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 280–285.

[27] H. Noguchi, K. Ikegami, K. Kushida, K. Abe, S. Itai, S. Takaya, N. Shimomura, J. Ito, A. Kawasumi, H. Hara *et al.*, "7.5 a 3.3 ns-access-time 71.2 μw/mhz 1mb embedded stt-mram using physically eliminated read-disturb scheme and normally-off memory architecture," in *2015 IEEE International Solid-State Circuits Conference-(ISSCC) Digest of Technical Papers*. IEEE, 2015, pp. 1–3.

[28] S. Patil, A. Lyle, J. Harms, D. J. Lilja, and J.-P. Wang, "Spintronic logic gates for spintronic data using magnetic tunnel junctions," in *2010 IEEE International Conference on Computer Design*. IEEE, 2010.

[29] S. Resch, S. K. Khatamifard, Z. I. Chowdhury, M. Zabihi, Z. Zhao, H. Cilasun, J.-P. Wang, S. S. Sapatnekar, and U. R. Karpuzcu, "Mouse: Inference in non-volatile memory for energy harvesting applications," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 400–414.

[30] S. Resch, S. K. Khatamifard, Z. I. Chowdhury, M. Zabihi, Z. Zhao, J.-P. Wang, S. S. Sapatnekar, and U. R. Karpuzcu, "Pimball: Binary neural networks in spintronic memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1–26, 2019.

[31] D. Saida, S. Kashiwada, M. Yakabe, T. Daibou, N. Hase, M. Fukumoto, S. Miwa, Y. Suzuki, H. Noguchi, S. Fujita *et al.*, "Sub-3 ns pulse with sub-100 μa switching of 1x–2x nm perpendicular mtj for high-performance embedded stt-mram towards sub-20 nm cmos," in *2016 IEEE Symposium on VLSI Technology*. IEEE, 2016, pp. 1–2.

[32] 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 *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 273–287.

[33] N. Talati *et al.*, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, 2016.

[34] P. L. Thangkhiew *et al.*, "Efficient mapping of boolean functions to memristor crossbar using magic nor gates," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 8, 2018.

[35] Y. Tomita and K. M. Svore, "Low-distance surface codes under realistic quantum noise," *Physical Review A*, vol. 90, no. 6, p. 062320, 2014.

[36] W. J. Townsend, E. E. Swartzlander Jr, and J. A. Abraham, "A comparison of dadda and wallace multiplier delays," in *Advanced signal processing algorithms, architectures, and implementations XIII*, vol. 5205. International Society for Optics and Photonics, 2003.

[37] S. Umesh and S. Mittal, "A survey of spintronic architectures for processing-in-memory and neural networks," *Journal of Systems Architecture*, vol. 97, pp. 349–372, 2019.

[38] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[39] N. K. Upadhyay *et al.*, "Engineering tunneling selector to achieve high non-linearity for 1s1r integration," *Frontiers in Nanotechnology*, 2021.

[40] J.-P. Wang, S. S. Sapatnekar, C. H. Kim, P. Crowell, S. Koester, S. Datta, K. Roy, A. Raghunathan, X. S. Hu, M. Niemier *et al.*, "A pathway to enable exponential scaling for the beyond-cmos era," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[41] 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, 2018.

[42] 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.

[43] Y. Zhang *et al.*, "Vertically integrated zno-based 1d1r structure for resistive switching," *Journal of Physics D: Applied Physics*, vol. 46, no. 14, p. 145101, 2013.