Path-based Processing using In-Memory Systolic Arrays for Accelerating Data-Intensive Applications

Muhammad Rashedul Haq Rashed*, Sven Thijssen†, Sumit Kumar Jha‡, Hao Zheng*, and Rickard Ewetz*
*Department of Electrical and Computer Engineering, University of Central Florida, Orlando, USA
†Department of Computer Science, University of Central Florida, Orlando, USA
‡Computer Science Department, Florida International University, Miami, USA
{muhammad.rashed, sven.thijssen, hao.zheng, rickard.ewetz}@ucf.edu, jha@cs.fiu.edu

***Abstract*—The next wave of scientific discovery is predicated on unleashing beyond-exascale simulation capabilities using in-memory computing. Path-based computing is a promising in-memory logic style for accelerating Boolean logic with deterministic precision. However, existing studies on path-based computing are limited to executing small combinational circuits. In this paper, we propose a framework called PSYS to accelerate data-intensive scientific computing applications using path-based in-memory systolic arrays. The approach leverages path-based computing for multiplying known constants with an unknown operand, which substantially reduces the computational complexity compared with general purpose multiplication of two unknown operands. The systolic arrays minimize data movement by storing the matrix elements using non-volatile memory and performing processing in-place. The framework decomposes unstructured computations to the systolic arrays while considering the non-regular computational patterns of the applications. Our experimental evaluations employ applications from the domains of engineering, physics, and mathematics. The experimental results demonstrate that compared with the state-of-the-art, the PSYS framework improves energy and latency by a factor of 101x and 23x, respectively.**

2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD) | 979-8-3503-2225-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAD57390.2023.10323622

I. Introduction

The availability of digital data has fueled the growth of data-centric applications within computer vision [1], social networks [2], and scientific computing [3]. The forthcoming era of scientific exploration hinges on the expansion of scientific simulations [4]. Numerous intricate systems are replicated through digital twins [5], and simulations are employed for intelligent decision-making processes [6]. However, current high-performance computing systems face significant challenges in handling simulations beyond the exascale level, primarily due to the separation of memory and compute units in the traditional von Neumann architecture [7]. In the relentless pursuit of faster and more efficient computational systems, the industry is witnessing a paradigm shift towards a more effective computing model. This has sparked investigations into alternative computing technologies and paradigms such as quantum computing [8], optical computing [9], and in-memory computing [10]. One notable area of interest in recent years has been processing in-memory using emerging non-volatile memories, which has garnered substantial attention due to its ability to achieve energy-efficient in-situ processing [11].

This work was in part supported by NSF awards # 2319399, # 2113307, the University of Central Florida, and Texas STARs award to Sumit Jha.

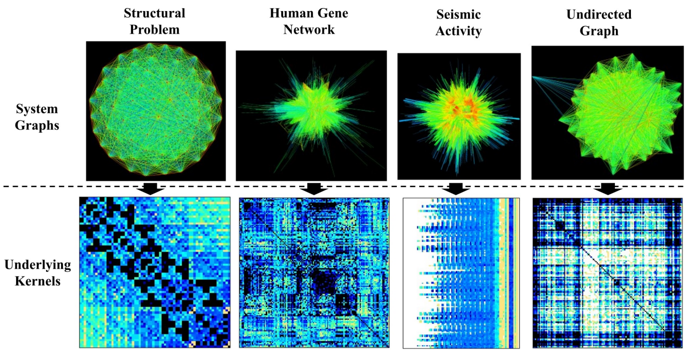


Fig. 1: Underlying matrix-vector multiplication kernels of four scientific simulation problems from SuiteSparse [12].

In-memory computing has been investigated using logic styles such as analog matrix-vector multiplication [13], MAGIC [14], IMPLY [15], streaming-based computing [16], flow-based computing [17–19], and path-based computing [20]. Different in-memory logic styles offer unique advantages and are subject to different limitations. While analog matrix-vector multiplication is extremely efficient, it lacks the deterministic precision provided by digital in-memory computing paradigms [21, 22]. The digital bit-wise-in-bulk paradigms MAGIC and IMPLY have been the foundation for several in-memory computing architectures [23–25]. The limitation is that the bit-wise parallelism requires each multiplication operation to be realized using a general purpose element-wise multiplication operation. This paper focuses on path-based computing that is capable of compiling arbitrary Boolean functions into compact look-up tables [20]. The path-based computation is (i) high-speed, (ii) extremely energy-efficient, (iii) in-situ processing, and (iv) deterministic precision [20]. While previous studies on path-based computing only focus on realizing small Boolean functions, this paper aims to accelerate full-blown scientific simulations using in-memory computing. An overview of the underlying matrix kernels of four scientific computing applications is shown in Figure 1. The main advantage of path-based computing is that it allows each element-wise multiplication with an operand and constant to be compiled into an abstract look-up table, which leads to substantial performance improvements compared with general purpose element-wise multiplication due to the reduction in computational complexity. On the other hand, the path-based paradigm requires careful architecture design to maximize parallelism and minimize data movement within the system architecture.

Various types of systolic arrays have been proposed for the acceleration of data-intensive applications [26, 27]. The systolic array minimize data movement by keeping different operands in-place and moving others systematically through the architecture. Google's tensor processing unit (TPU) was for example based on systolic arrays that store matrix-values in-place and pipeline the vector operands through the architecture [28]. Systolic arrays have been used to accelerate matrix-vector multiplication operations that are larger than the maximum crossbar dimensions in [29]. The limitation of systolic arrays is that they are customized for dense matrix-vector, or matrix-matrix multiplication operations. The hardware utilization quickly becomes prohibitively low if the matrix kernel is sparse. Various blocking and decomposition schemes have been proposed to improve the array utilization [30–33]. The use of reconfigurable adder trees have been demonstrated useful for sparse matrix representations [34, 35].

In this paper, we propose a framework called PSYS to accelerate data-intensive scientific computing applications using path-based in-memory systolic arrays. Path-based computing is used to efficiently accelerate element-wise multiplications of operands and constants using compact look-up tables. The LUTs realized using resistive random access memory (RRAM) are combined with CMOS based adders to form multiply-and-accumulate units within a systolic array. The systolic array is designed using bit-slicing decomposition schemes to maximize computational efficiency. Unstructured computations are decomposed into the systolic arrays while considering the non-regular computational patterns of the applications, with the objective of maximizing the utilization of the systolic arrays.

The main contributions of the paper are summarized, as follows:

- Leverage path-based in-memory computing to realize configurable LUTs. The LUTs realize constant and operand multiplication instead of general purpose multiplication of a pair of operands.
- The systolic arrays perform in-situ computation of dense matrix-vector operation. The arrays leverage a hybrid of crossbars and CMOS logic, with shift-and-add decomposition for improved efficiency.
- The non-regular computational patterns are decomposed to regular computational patterns. The approach maximizes the utilization of the systolic arrays.
- The PSYS framework is evaluated using 20 applications from the domains of engineering, physics, and mathematics. Compared with the state-of-the-art, the experimental evaluation demonstrate energy and latency improvements of 101x and 23x, respectively.

The remainder of the paper is organized as follows: preliminaries and the motivation are presented in Section II. The PSYS framework is introduced in Section III. The synthesis methodology of the framework is explained in Section IV. Decomposition method of unstructured system is discussed in Section V. The architecture and the experimental evaluations are discussed in Section VI. The paper is concluded in Section VII.

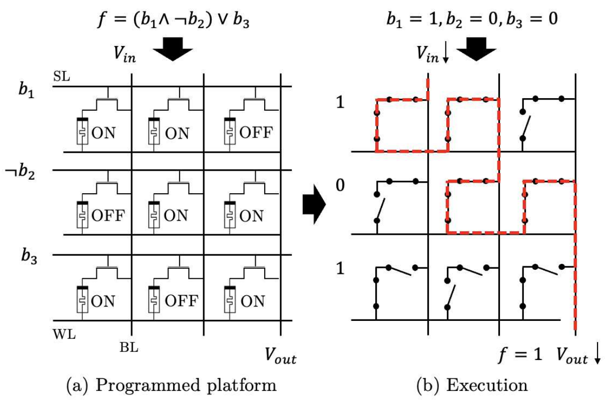


Fig. 2: Path-based in-memory computing using 1T1M crossbar. (a) Programming of the crossbar to perform Boolean function $f(b_1, b_2, b_3)$, and (b) execution of f for [1,0,0] instances of variables (b_1, b_2, b_3). (WL=wordline, BL=bitline and SL=selector line.)

II. Preliminaries

In this section, we first explain the operating principles of Path-based in-memory computing. Next, we describe the design of systolic array to accelerate MVM operations. Finally, we discuss the motivation for designing path-based in-memory systolic array architectures.

A. *Path-based computing*

Path-based computing is a READ-based digital in-memory computing paradigm on 1T1M (one-transistor one-memristor) crossbars [20]. The computing paradigm is READ-based because it only requires the memristors to be programmed once to a resistive state. Path-based computing consist of two steps. First, a Boolean function is synthesized into a crossbar design, which is subsequently mapped to a programmed platform. For example, in Figure 2, the Boolean function $f = (b_1 \wedge \neg b_2) \vee b_3$ is synthesized into the programmed platform in Figure 2(a). In a programmed platform, the input variables are assigned to the selector-lines, and the memristors are programmed to either a low (ON) or high resistive state (OFF). The next step is the execution where the Boolean function is evaluated for a given input vector. In Figure 2(b), we are provided the input vector $b_1 = 1$, $b_2 = 0$, and $b_3 = 0$. Based on these values, the corresponding selector lines are charged accordingly. For example, when $b_1 = 1$, the transistors connected to this selector line are closed. For $b_3 = 0$, the selector line is open. Next, a high input voltage is applied to the input bit line V_{in}. This will result in an electrical current flowing through the crossbar. If the electrical current from the input bit line to the output bit line through memristors in the low resistive state, we conclude that the function evaluates to true. Otherwise, if there is no such path from input to output, then the function evaluates to false. In Figure 2(b), we observe that there is a path from input to output, and thus f evaluates to true.

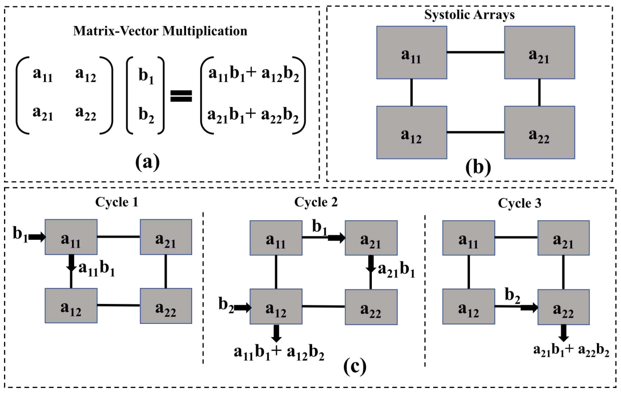
B. Systolic Arrays

Systolic arrays are powerful parallel processors that can accelerate matrix-vector multiplications (MVM) [36–38]. To perform MVM operations within traditional architecture, iterative data fetch from the main memory is required to bring the MVM operands to the processor. This results in a scenario where a piece of data may get repeatedly fetched from and sent back to the main memory which imposes significant load on the valuable data transfer bandwidth. On the contrary, the systolic array processors are designed in a way that when a piece of data for MVM is fetched from the main memory, the utilization of the fetched data is maximized before it is sent back to the memory. We explain the concept of MVM operation using systolic arrays in Figure 3.

In Figure 3(a), we show a 2×2 MVM operation. The outputs of the MVM is achieved by sequential multiplication and accumulation of operands. In Figure 3(b), we show the assignment of the matrix operands into an architecture of orthogonal systolic arrays [39]. Note that the matrix operands are mapped in a *transposed* manner within the arrays. We illustrate the MVM workflow of the systolic arrays in Figure 3(c). Each of the arrays perform a multiplication and accumulation operation. The computation within an array can be generalized as follows:

$$\begin{aligned} & out_j = (a_{ij} \times b_j) + out_{j-1} \\ \text{s.t.} \quad & out_{j-1} = a_{i(j-1)} \times b_{j-1}, \;\; out_0 = 0, \;\; \forall (i,j) \in Z^+ \end{aligned}$$

For Cycle 1 in Figure 3(c), the top-left systolic array performs a multiplication on the stored a_{11} matrix operand with incoming input-vector operand b_1. In the following cycle, the operand b_1 is routed horizontally to the systolic array storing a_{21}. At the same time, operand b_2 is introduced on the array storing a_{12} in similar data-flow direction. In an orthogonal direction of the data (b_i) flow, the multiplication results $a_{ij} \times b_j$ are routed. Within this data-flow, the input operands (a_{ij}, b_j) and the output results are not returned to the main memory until the entire MVM operation is concluded.



Matrix-Vector Multiplication
$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_{11}b_1 + a_{12}b_2 \\ a_{21}b_1 + a_{22}b_2 \end{pmatrix}$
(a)
Systolic Arrays
a_{11}
a_{21}
a_{12}
a_{22}
(b)
Cycle 1
Cycle 2
Cycle 3
b_1
$a_{11}b_1$
$a_{21}b_1$
b_2
$a_{11}b_1 + a_{12}b_2$
$a_{21}b_1 + a_{22}b_2$
(c)

Fig. 3: Matrix-vector multiplication (MVM) using systolic arrays. (a) A 2×2 matrix-vector multiplication, (b) transposed matrix weight mapping in systolic arrays and, (c) workflow of MVM within systolic arrays.

C. Motivation

In this section, we discuss the motivation behind designing a path-based in-memory systolic array architecture.

Although traditional systolic array architectures highly parallelize the MVM operation and reduce the repeated access of same data from memory, they are still required to fetch both the matrix and vector data from the memory. This large amount of data movement requires localized buffers within the array units which are costly in terms of energy and area [40–42]. Large systolic array architectures, such as Google TPU employ embedded memory units to reduce off-chip data communications [36]. However, these embedded memory are power hungry and they create a localized *memory-wall* bottleneck due to the performance gap between the memories and the processors [43]. In-memory computing can be a promising solution to overcome this memory wall by in-situ processing of computational operands.

Some recent works have explored the design of in-memory systolic arrays [29, 44]. However, these works are based on analog in-memory computing processors. Previous studies have shown that analog in-memory computing is vulnerable to various errors that result in limited precision computation [21, 22, 45]. Therefore, an in-memory systolic array capable of performing computation with deterministic precision is still missing.

Path-based computing is a natural candidate for an in-memory systolic array system due to its inherent streaming-style computing capability and its ability to perform deterministic precision computation. Additionally, for fixed-weight matrix operands, the path based in-memory computing only fetches the input-vector operands and thus reduces memory fetch complexity from $O(N^2)$ to $O(N)$.

III. The PSYS Framework

In this section, we introduce the PSYS framework. An architectural overview of PSYS is illustrated in Figure 4.

The overall architecture consists of a CPU unit, a DRAM memory unit and a series of systolic array based processing elements (PEs) interconnected with high-speed bus. The systolic array arrangement within a PE is shown on the top-middle of the figure. Each of the array units within the

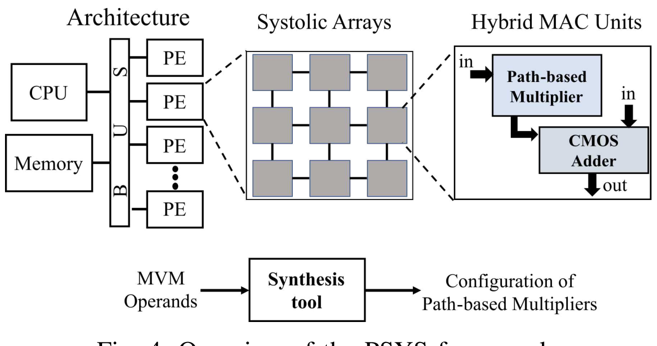


Fig. 4: Overview of the PSYS framework.

PEs consists of a hybrid multiply and accumulate (MAC) unit which is shown in the top-right of the figure. A path-based ITIM crossbar architecture constitute the multiplier unit. Multiplication operations are converted into kernels for path-based computing and are bound to the crossbar. The output of the multiplier unit is fed into a CMOS adder unit. The adder also receives inputs from the previous systolic array unit. The output of the MAC operation is collected from the output side of the adder unit.

The overview of the synthesis operation is shown at the bottom of Figure 4. The input to the synthesis tool is the operands of the MVM operation. The output of the tool is the configurations of the path-based multiplier units. The synthesis process is described in the Section IV.

IV. Synthesis

In this section, we explain the synthesis flow for the PSYS framework. An overview of the synthesis flow is illustrated in Figure 5.

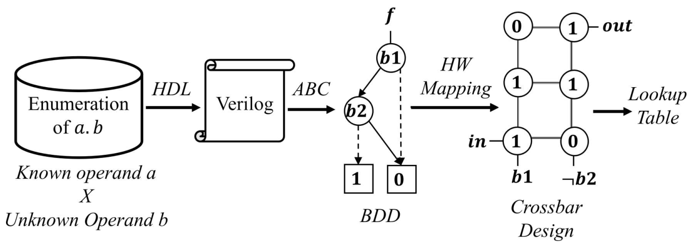


Fig. 5: Synthesis flow for PSYS framework.

The synthesis process starts with an exhaustive enumeration of multiplications of a p-bit known number a with a p-bit unknown number b. The final output of the synthesis process is a lookup table of crossbar designs for all the enumerated multiplications. First, the multiplication operations are converted into hardware description language (HDL) verilog. Next, ABC [46] tool is used to convert the verilog files into binary decision diagram (BDD) netlists. Finally, BDDs are synthesized into path-based crossbar designs using the PATH tool [20].

The synthesis problem consists of two major challenges:

1) **Scaling**: Our experimental evaluation shows that the path-based design scale poorly for high precision multiplication operation. In particular, the netlist size tend to explode for high-precision of the unknown operand b. However, the impact of the higher-precision of the known operand a is not very significant as its a constant value.
2) **Enumeration**: The exhaustive enumeration of multiplication operations involve the enumeration of all possible bit-patterns of the known operand. This translates into an enumeration complexity of $O(2^p)$.

These two challenges are co-related. We solve the challenges using an *adaptive bit-slicing* scheme.

A. Problem Formulation

To address the first challenge, the aim is to find a bit-slicing width of the unknown operand b that minimizes the normalized cost of the multiplication. Let m denote the bit-slicing precision of b. To address the second challenge, we aim to maximize the bit-slicing precision n of the known operand a while keeping the enumeration complexity within an acceptable limit. This can be formalized, as follows:

$$\min_{m} \max_{n} \quad cost(S_i).cost(T_j), \qquad \forall i,j \in Z^+ \tag{1}$$

where S_i is the semi-perimeter (number of rows+column) of the crossbar design for unknown operand precision of i and T_j is the synthesis runtime for the exhaustively enumerated multiplication operations for known operand precision of j.

B. Adaptive Bit-slicing

The concept of adaptive bit-slicing is explained with an example in Figure 6.

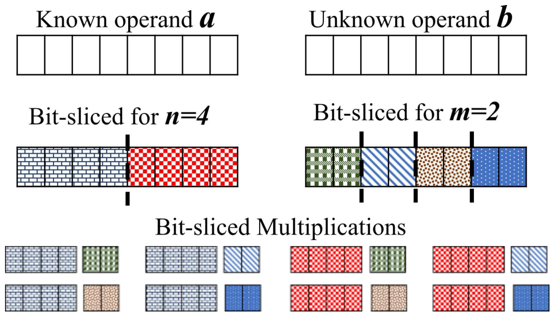


Fig. 6: Adaptive bit-slicing of MVM operands.

On the top of the figure, we show two 8-bit operands a and b where the value of a is known. Next, bit-slicing of a and b with different bit-slicing width is shown in the middle of the figure. For the known and the unknown operands, bit-slicing width of 4-bit and 2-bit, respectively, are selected. As a result of the bit-slicing, the original multiplication operation is now partitioned into a series of multiplication and accumulation operations. All the different multiplicand pairs are shown at the bottom of the figure. During accumulation of the partitioned multiplications, shifting operations are required to align the multiplication results to their corresponding weight locations. The entire multiplication operation can be expressed as follows:

$$mult = \sum_{i=1}^{p/n} \sum_{j=1}^{p/m} a_i \times b_j \times 2^{(x_i+y_j-2)}, \quad \forall (i,j) \in Z^+$$

Here, p is the original bit-precision of the operands, (a_i, b_j) are a pair of bit-sliced multiplicands, the LSB of a_i is the x_i^{th} LSB of a and the LSB of b_j is the y_j^{th} LSB of b. For alignment, each partial multiplication of a_i & b_j is shifted by $(x_i + y_j - 2)$ bit-position to the left.

V. Decomposing Unstructured Computation

In this section, we develop a matrix compression algorithm to decompose unstructured sparse matrices to PSYS framework.

In Figure 7(a), we show the sparsity pattern of a system *Si*2 from the SuiteSparse matrix collection [12]. It can be

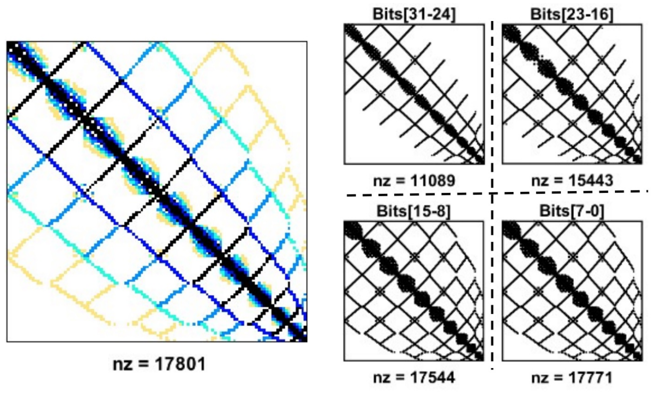


Fig. 7: Bit-slicing of matrix operands. (a) Sparsity pattern of the system *Si2* from quantum chemistry domain [12], and (b) bit-sliced matrices *Si2* for 8-bit slicing width. *nz* denotes the number of non-zeros within the matrix.

observed that the matrix is very sparse and a naive mapping of matrix operands into crossbars will result in significant under-utilization of hardware. Also, the sparsity pattern shows that the distribution of operand weight is not uniform. The more darker (lighter) parts of the sparsity pattern denote higher (lower) valued operands. This results in that, when we bit-slice the matrix operands, it generates several matrices with different sparsity patterns. For instance, Figure 7(b) shows the conversion of 32-bit $Si2$ matrix into four sparse matrices, each with 8-bit operands. We target to develop an algorithm to automatically decomposed the bit-sliced matrices that can be efficiently evaluated within the PSYS framework.

Recent works have proposed a blocked-shifting based compression of sparse matrices to decompose the original matrix into a series of denser matrix blocks [47]. The concept is illustrated with an example in Figure 8. Figure 8(a) shows the bit-sliced matrix of $Si2$ system for the 8 most significant bits (31:24). In the first step, the matrix is partitioned into a number of rectangular blocks with a fixed number of rows as shown in Figure 8(a). For the remainder of this paper, we will use the term *block-size* to denote the number of matrix rows within the blocks. Next, each column within a block that contains a non-zero operand is selected and the zero values of

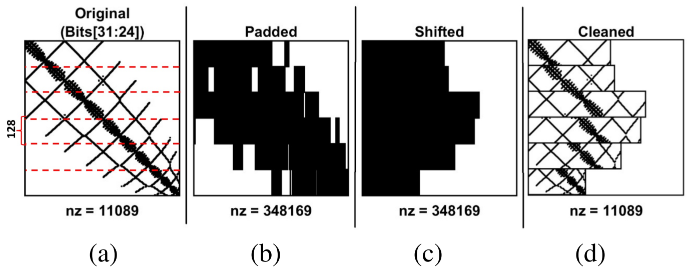


Fig. 8: Shifted compression of sparse matrix for fixed block-size of 128 [47]. (a) Original matrix, (b) padding of non-zero columns, (c) shifting, and (d) cleaning of padded operands.

the selected columns are padded with dummy values. Next, the non-zero columns within each blocks are shifted to the left. The padding and shifting steps are shown in Figure 8(b) and 8(c), respectively. Finally, the dummy values are cleared and a set of relatively denser matrix blocks are achieved.

While this algorithm works for a few of the sparse systems, it cannot guarantee dense partitioning for all sparse systems. It is evident form Figure 8(d) that the blocks extracted by the algorithm are not significantly dense. This is due to that the density of the compressed blocks is highly dependent on the localized data-pattern and the selected *block-size*. However, the algorithm fixes a *block-size* for all matrices irrespective of the data-arrangement of the system. We have already observed in Figure 7 that each of bit-sliced matrices may have different sparsity patterns. Therefore, the block dimension should be dynamically updated for the target matrices. We present a case study on the impact of using different block sizes for the matrix(31:24) of $Si2$ in Figure 9. Figure 9(a)$-$(d) shows the blocked matrices for *block-size* of 128, 64, 32 and 16, respectively. The Figure shows that for smaller *block-size*, the extracted matrix blocks become increasingly denser.

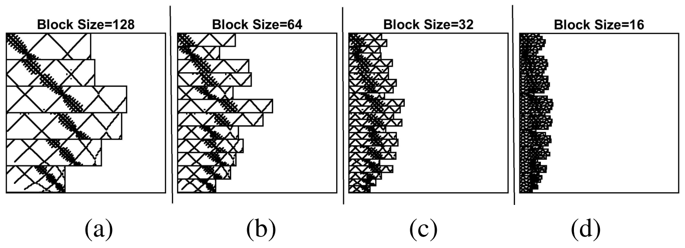


Fig. 9: Matrix Compression of matrix[31:24] of $Si2$ for variable block sizes of (a) 128, (b) 64, (c) 32, and (d) 16.

However, an exhaustive exploration of block sizes will pose two challenges:

1) **Complexity:** The computational complexity of block extraction will become $O(MN)$.
2) **Memoization:** While extracting the blocks, we are required to record (#*rows*/*block-size*) subsets of the input vector indices. For very small *block-size*, this bookkeeping can become expensive.

To overcome these challenges, we present Algorithm 1 where the block search complexity is $O(log_2(N))$. We also set a density threshold to run the algorithm until an acceptable density is achieved. The inputs to the algorithm are the original matrix A, matrix bit-slicing width n and the block density threshold d. The algorithm returns a library ($\mathcal{L}$) of denser matrix blocks, and the lookup table ($\mathcal{I}$) of the corresponding input vector subset indices. First, the algorithm decomposes the original matrix into bit-sliced matrices of n-bit width. Next, for each of the bit-sliced matrices, an iterative exploration of *block-size* is performed. In each subsequent iteration, the *block-size* is halved from its previous value, bringing the search complexity to $O(log_2(N))$. For each exploration of blocking, the blocks are compressed using the algorithm in Figure 8. For all the compressed blocks, the

```
Algorithm 1: Decomposing Sparse Matrices
Inputs: Matrix, $A$; bit-slicing width, $n$; block density
  threshold, $d$
Output: Compressed matrix block library, $\mathcal{L}$; look-up
  table of input-vector subset indices, $\mathcal{I}$
$\mathcal{L}, \mathcal{I} \leftarrow \phi$; \\initializing
$[r,c] \leftarrow size(A)$; \\matrix dimensions
$\mathcal{A} \leftarrow Bitslice(A,n)$; \\Bitslicing original matrix
for $all\ A_i \in \mathcal{A}$ do
    $block\_size \leftarrow r$; \\initial block size
    $\acute{d} \leftarrow density(A_i)$; \\initial density
    while $\acute{d} < d$ do
        $block\_size \leftarrow block\_size/2$; \\downsizing block
        $\{\mathcal{A}_{\lceil}, \acute{\mathcal{I}}\} \leftarrow Compress(Blocking(A_i, block\_size))$;
        \\matrix compression. $\mathcal{A}_{\lceil}$ stores compressed
          blocks, $\acute{\mathcal{I}}$ tracks corresponding column indices
        for $all\ compressed\ blocks\ A'_j \in \mathcal{A}_{\lceil}$ do
            $\acute{d} \leftarrow avg\_density(A'_j)$; \\running average
        end
        if ( $\acute{d} \geq d$) then
            $\mathcal{L} \leftarrow \mathcal{L} \cup (\mathcal{A}_{\lceil}, i)$; \\indexed matrix blocks
            $\mathcal{I} \leftarrow \mathcal{I} \cup (\acute{\mathcal{I}}, i)$; \\indices look-up table
        end
    end
end
return $\mathcal{L}, \mathcal{I}$;
```

running average density $\acute{d}$ is calculated. When the condition $\acute{d} \geq d$ is met, the block downsizing is stopped and the $\mathcal{L}$ and $\mathcal{I}$ are updated with current matrix blocks and corresponding input-vector indices, respectively.

The workflow of decomposition of sparse matrices for PSYS framework is illustrated in Figure 10. The figure shows how the original matrix is first bit-sliced into several matrices with different sparsity. Next, for each of the sliced matrices, compression is performed using different *block-sizes*. Next, the compressed blocks are transposed to align with the operand mapping flow of the systolic arrays. Finally, the transposed matrix blocks are hardware bound to the NVM systolic arrays using the look-up table of Section IV.

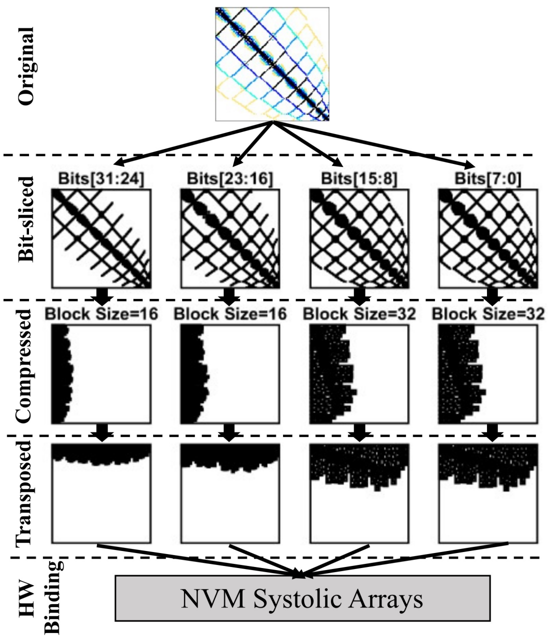


Fig. 10: Sparse matrix conversion using Algorithm 1.

VI. Experimental Evaluation

In this section, we experimentally evaluate the performance of the PSYS framework. We use an octa-core machine with 3.60 GHz Intel Core i9 processor with NVIDIA RTX 2070 and 64 GB RAM to conduct the experiments. We use a blend of MATLAB and Python scripts to decompose the sparse matrices and to generate the HDL codes. For synthesizing the path-based computing kernels, we use the ABC tool [46] and PATH [20] tool. For the evaluation of CMOS-based architectural components, we use the Synopsis Design Compiler tool. We couple the compiler with a *gscl*-45 nm technology library [48] to evaluate the power, area and latency overheads. To estimate the cross-architecture data-transfer cost, we utilize the the CACTI 7 tool [49] on 45 nm technology node. The power and area performance of the NVM crossbars are adapted from [50–52]. The operating latency of NVM devices are obtained from [53].

We first present the architecture overview of the PSYS framework in Section VI-A. Next, we evaluate the bit-slicing step of the synthesis flow in Section VI-B. Subsequently, we experimentally justify the NVM-CMOS hybrid design for the MAC unit in Section VI-C. Finally, we evaluate the performance of the PSYS framework for 20 sparse matrices from different domains of scientific computation in Section VI-D.

A. Architecture

In this section, we present the architecture of the processing elements (PEs) within the PSYS framework. The components of the PE architecture are illustrated in Figure 11.

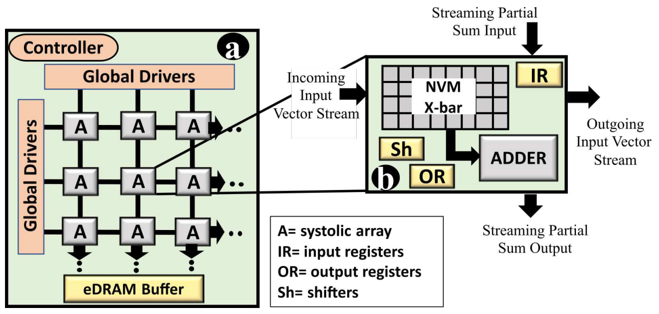


Fig. 11: PSYS PE architecture.

The overview of the PE architecture is shown in Figure 11(a). The PE consists of a collection of orthogonal systolic arrays (A). Each of the arrays perform multiply and accumulate (MAC) operation to process MVM applications. Controller and global drivers are used to program the arrays based on synthesized MVM operands. The components within a systolic array is shown in Figure 11(b). Each array is equipped with a NVM crossbar to perform multiplication and a CMOS adder to accumulate the results. The figure shows the incoming and the outgoing data flows within the array unit. Each array within the same row of arrays share the same set of input vectors. The vector is streamed side-wise throughout the PE. After the multiplication within an array is processed, the vector is streamed to the next row-parallel array. Simultaneously, partial sums of the MVM operations are streamed column-wise throughout the arrays. Each array within a PE column receive a stream of running partial sum from the previous array on top and streams the updated partial sum downwards to the next array in the same column. Each NVM crossbar performs several bit-sliced multiplication operations. Input registers (IR) and output registers (OR) are used to hold the input and output operands to enable pipelining of bit-sliced computations. Shifters (Sh) are used to perform shifting of bit-slices. The per-unit area-power costs of the systolic array components are listed in Table I. The dimensions for the NVM crossbars are selected to be 128×256. This rectangular shape is deliberately selected as the crossbar design of the bit-sliced multiplications assume a rectangular shape. This is experimentally demonstrated in the next subsection.

TABLE I: Area-Power Cost of Systolic Array Components

Component	Parameter	Specs	Area	Power
Crossbar	size	128×256	50 μm^2	0.6 mW
Adder	resolution	16 b	186.31 μm^2	0.08 mW
IR	size	10 B	10.5 μm^2	0.006 mW
OR	size	2 B	5.86 μm^2	0.003 mW
Sh	# unit	1	23.43 μm^2	0.02 mW
Local Bus	#wires	64	0.0075 mm^2	0.58 mW

B. Evaluation of Adaptive Bit-Slicing

In this section, we evaluate the adaptive bit-slicing step of synthesis. The step begins with an exhaustive enumeration of all possible multiplication operations of an unknown operand with a known operand of p-bit. Due to the $O(2^p)$ order of enumeration complexity, we make an engineering choice to fix the bit-slicing width of the known operand to 8-bits. This helps us to limit the enumeration size to 256 different multiplication operations. Next, we generate the look-up table of crossbar designs for different bit-slicing width of the unknown operand. Each of the look-up table for different bit-slicing width (1, 2, 4, etc.) contain 256 entries. Next, we evaluate the worst-case semi-perimeter for each of the table. The results of this experiment are presented in Table II. The table presents the total semi-parameter overhead to perform 8-bit multiplication (8-bit known operand $\times$ 8-bit unknown operand) for different bit-slicing width of the unknown operand. The table shows that the overhead is least when the unknown operand is sliced using a bit-width of 2-bits. It can also be observed that the optimum crossbar design is rectangular (24×48). Based on this observation, we select rectangular crossbars for the design the systolic arrays within the PEs.

TABLE II: Overhead Comparison for Different Bit-slicing.

Worst-case Overhead for 8-bit Multiplication	Bit-slicing Width of Unknown Operand 1-bit	2-bit	4-bit	8-bit
#wordlines	8	24	100	414
#bitlines	72	48	84	327
Semi-perimeter	80	**72**	184	741

C. Evaluation of MAC Unit

In this section, we perform a comparative performance evaluation of the proposed hybrid multiply-accumulate (MAC) design. For the comparative study, we consider three different MAC designs as shown in Figure 12. Figure 12(a)-(c) show a purely CMOS-based MAC design, a purely NVM-based MAC deign and the proposed hybrid MAC design, respectively.

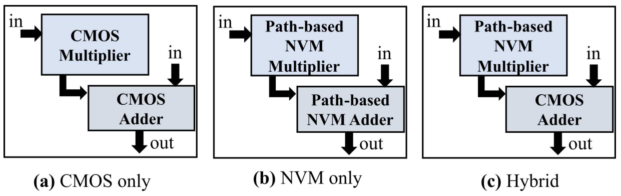
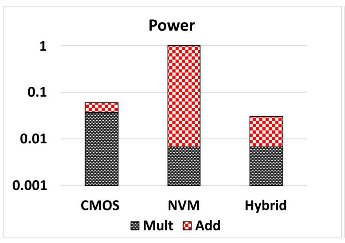


Fig. 12: MAC architectures for comparative evaluations.

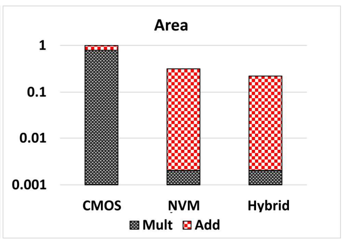
We present the comparative power and area performance of the three MAC architectures in Figure 13(a) and (b). The figures show that, CMOS-based multiplier are more power hungry and incur more area overhead than the NVM-based multiplier. On the other hand, the NVM adder is more expensive in terms of power and area than the CMOS adder. This is due to the fact that the adders in the MAC architecture are general purpose and the path-based computing systems scale poorly for general purpose operations. It can be observed from the figures that, the hybrid MAC architecture outperforms both the CMOS-only and NVM-only MAC architectures in terms of power and area overhead. The evaluation shows that, compared to the CMOS-only and NVM-only MAC architectures, the hybrid MAC is 50% and 97% more power efficient, respectively. Also, the hybrid MAC reduces area overhead by 78% and 30%, respectively, over the CMOS-only and NVM-only MAC architectures.

D. Evaluation with MVM Applications

In this section, we evaluate the performance of the PSYS framework on 20 applications from different domains of scientific simulation. The selected applications are listed in



(a)



(b)

Fig. 13: (a) Power, and (b) area comparison for CMOS-only, NVM-only and hybrid MAC architectures.

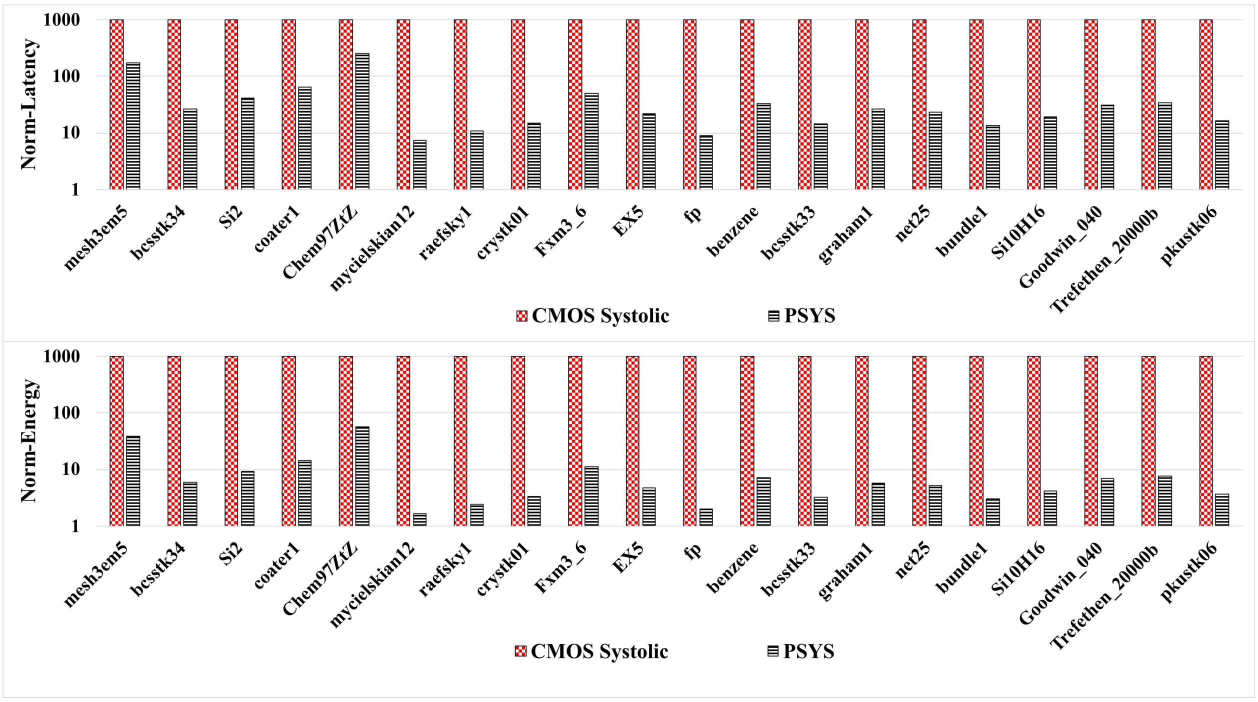


Fig. 14: Comparison of latency, and energy for traditional CMOS-based systolic array architecture and the PSYS framework on twenty benchmarks of the SuiteSparse Matrix Collection [12].

TABLE III: Overview of benchmarks from the SuiteSparse Matrix Collection [12].

Applications	Systems	Matrix Dimensions	#Non-zeros
mesh3em5	Structural	289×289	1377
bcsstk34	Structural	588×588	21418
Si2	Quantum Chemistry	769×769	17801
coater1	Fluid Dynamics	1348×1348	19457
Chem97ZtZ	Mathematical	2541×2541	7361
mycielskian12	Undirected Graph	3071×3071	407200
raefsky1	Fluid Dynamics	3242×3242	293409
crystk01	Materials	4875×4875	315891
fxm3_6	Optimization	5026×5026	94026
EX5	Combinatorial	6545×6545	295680
fp	Electromagnetics	7548×7548	834222
benzene	Quantum Chemistry	8219×8219	242669
bcsstk33	Structural	8738×8738	591904
graham1	Fluid Dynamics	9035×9035	335472
net25	Optimization	9520×9520	401200
bundle1	Computer Vision	10581×10581	770811
Si10H16	Quantum Chemistry	17077×17077	875923
Goodwin_040	Fluid Dynamics	17922×17922	561677
Trefethen_20000b	Combinatorial	19999×19999	554435
pkustk06	Structural	43164×43164	2571768

Table III. The systems are very sparse with an average data density of only 1.4%. The systems are decomposed into denser matrix blocks using the Algorithm 1. We select a reasonable density threshold of $d = 60\%$ for the system conversion. The performance of the PSYS is compared with traditional CMOS-based systolic architecture. The comparative latency and energy performance are illustrated in Figure 14.

The results show that the PSYS framework achieves 23x speedup on average compared to the traditional systolic array architecture. This remarkable improvement is the result of limited data-movement within the PSYS framework due to that it only streams the input vectors. On the contrary, traditional architecture streams both the system matrix and the input vector elements, making the data movement complexity in the order of $O(N^2)$. The PSYS system also achieves 101x more energy-efficiency compared to the traditional architecture. As shown in the previous subsection, the hybrid MAC units of the PSYS framework are more power efficient than the traditional MAC units. This improvement is amplified by the speedup achieved by limited data movement, bringing down the total energy consumption significantly.

VII. Conclusion

In this paper, we have introduced PSYS, a novel framework that significantly accelerates data-intensive scientific computing applications by leveraging path-based in-memory systolic arrays. The core of the PSYS approach lies in its application of path-based computing, which facilitates efficient multiplication between known constants and unknown operands, thus mitigating the inherent computational complexity of traditional multiplication methods involving two unknown operands. Additionally, the systolic arrays in the PSYS framework store matrix elements using non-volatile memory and conduct in-place processing, drastically minimizing data movement. One of the strengths of PSYS is its capacity to convert unstructured computations into forms compatible with systolic arrays, exploiting the unique non-regular computational patterns of the applications being accelerated. In future research, we intend to explore the use of machine learning methods for accelerating the PSYS framework by avoiding the algorithmic construction of libraries for matrix-vector multiplications. Instead, we will seek to search and reason about the library by storing it in a compact neural network representation.

References

[1] A. Voulodimos, N. Doulamis, A. Doulamis, E. Protopapadakis, *et al.*, "Deep learning for computer vision: A brief review," *Computational intelligence and neuroscience*, vol. 2018, 2018.

[2] T. A. Snijders, "Statistical models for social networks," *Annual Review of Sociology*, vol. 37, no. 1, pp. 131–153, 2011.

[3] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. Chue Hong, M. Davis, R. T. Guy, S. H. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, *et al.*, "Best practices for scientific computing," *PLoS biology*, vol. 12, no. 1, p. e1001745, 2014.

[4] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, and K. Willcox, "Brochure on basic research needs for scientific machine learning: Core technologies for artificial intelligence,"

[5] S. Niederer, M. Sacks, M. Girolami, and K. Willcox, "Scaling digital twins from the artisanal to the industrial," *Nature Computational Science*, vol. 1, pp. 313–320, 05 2021.

[6] L. Himanen, A. Geurts, A. S. Foster, and P. Rinke, "Data-driven materials science: Status, challenges, and perspectives," *Advanced Science*, vol. 6, no. 21, p. 1900808, 2019.

[7] A. Jaiswal, I. Chakraborty, A. Agrawal, and K. Roy, "8t sram cell as a multibit dot-product engine for beyond von neumann computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2556–2567, 2019.

[8] E. National Academies of Sciences, Medicine, *et al.*, "Quantum computing: progress and prospects," 2019.

[9] C. Qian, X. Lin, X. Lin, J. Xu, Y. Sun, E. Li, B. Zhang, and H. Chen, "Performing optical logic operations by a diffractive neural network," *Light: Science & Applications*, vol. 9, no. 1, p. 59, 2020.

[10] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, pp. 529–544, 2020.

[11] A. Gebregiorgis, H. A. Du Nguyen, J. Yu, R. Bishnoi, M. Taouil, F. Catthoor, and S. Hamdioui, "A survey on memory-centric computer architectures," *J. Emerg. Technol. Comput. Syst.*, vol. 18, oct 2022.

[12] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[13] C. Li, M. Hu, Y. Li, H. Jiang, N. Ge, E. Montgomery, J. Zhang, W. Song, N. Dávila, C. E. Graves, *et al.*, "Analogue signal and image processing with large memristor crossbars," *Nature Electronics*, vol. 1, no. 1, p. 52, 2018.

[14] 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*, vol. 61, no. 11, pp. 895–899, 2014.

[15] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (imply) logic: Design principles and methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2054–2066, 2013.

[16] M. R. H. Rashed, S. Thijssen, S. K. Jha, F. Yao, and R. Ewetz, "Stream: Towards read-based in-memory computing for streaming based processing for data-intensive applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[17] D. Chakraborty and S. K. Jha, "Automated synthesis of compact crossbars for sneak-path based in-memory computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 770–775, IEEE, 2017.

[18] A. U. Hassen, D. Chakraborty, and S. K. Jha, "Free binary decision diagram-based synthesis of compact crossbars for in-memory computing," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 5, pp. 622–626, 2018.

[19] A. Velasquez and S. K. Jha, "Parallel boolean matrix multiplication in linear time using rectifying memristors," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1874–1877, IEEE, 2016.

[20] S. Thijssen, S. K. Jha, and R. Ewetz, "Path: Evaluation of boolean logic using path-based in-memory computing," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 1129–1134, 2022.

[21] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 476–488, IEEE, 2015.

[22] M. Le Gallo, A. Sebastian, R. Mathis, M. Manica, H. Giefers, T. Tuma, C. Bekas, A. Curioni, and E. Eleftheriou, "Mixed-precision in-memory computing," *Nature Electronics*, vol. 1, no. 4, pp. 246–253, 2018.

[23] 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, 2019.

[24] A. Zulehner, K. Datta, I. Sengupta, and R. Wille, "A staircase structure for scalable and efficient synthesis of memristor-aided logic," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 237–242, 2019.

[25] D. Bhattacharjee and A. Chattopadhyay, "Synthesis and technology mapping for in-memory computing," in *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*, pp. 317–353, Springer, 2022.

[26] E. J. Houtgast, V.-M. Sima, K. Bertels, and Z. Al-Ars, "An fpga-based systolic array to accelerate the bwa-mem genomic mapping algorithm," in *2015 international conference on embedded computer systems: Architectures, modeling, and simulation (samos)*, pp. 221–227, IEEE, 2015.

[27] G. Peng, L. Liu, S. Zhou, S. Yin, and S. Wei, "A 2.92-gb/s/w and 0.43-gb/s/mg flexible and scalable cgra-based baseband processor for massive mimo detection," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 2, pp. 505–519, 2019.

[28] J. J. Zhang, K. Basu, and S. Garg, "Fault-tolerant systolic array based accelerators for deep neural network execution," *IEEE Design & Test*, vol. 36, no. 5, pp. 44–53, 2019.

[29] N. Challapalle, S. Rampalli, M. Chandran, G. Kalsi, S. Subramoney, J. Sampson, and V. Narayanan, "Psb-rnn: A processing-in-memory systolic array architecture using block circulant matrices for recurrent neural networks," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 180–185, IEEE, 2020.

[30] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on parallel and distributed systems*, vol. 10, no. 7, pp. 673–693, 1999.

[31] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pp. 161–170, 2014.

[32] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, "An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on gpus," in *Proceedings of the 28th ACM international conference on Supercomputing*, pp. 273–282, 2014.

[33] X. He, S. Pal, A. Amarnath, S. Feng, D.-H. Park, A. Rovinski, H. Ye, Y. Chen, *et al.*, "Sparse-tpu: Adapting systolic arrays for sparse matrices," in *Proceedings of the 34th ACM international conference on supercomputing*, pp. 1–12, 2020.

[34] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pp. 63–74, 2005.

[35] J. Sun, G. Peterson, and O. Storaasli, "Sparse matrix-vector multiplication design on fpgas," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pp. 349–352, IEEE, 2007.

[36] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

[37] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, pp. 1–6, 2017.

[38] S. Kundu, S. Banerjee, A. Raha, S. Natarajan, and K. Basu, "Toward functional safety of systolic array-based deep learning hardware accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 3, pp. 485–498, 2021.

[39] H. T. Kung and C. E. Leiserson, "Systolic arrays (for vlsi)," in *Sparse Matrix Proceedings 1978*, vol. 1, pp. 256–282, Society for industrial and applied mathematics Philadelphia, PA, USA, 1979.

[40] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," in *2017 IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1–8, IEEE, 2017.

[41] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[42] H. Valavi, P. J. Ramadge, E. Nestler, and N. Verma, "A 64-tile 2.4-mb in-memory-computing cnn accelerator employing charge-domain compute," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 6, pp. 1789–1799, 2019.

[43] H. Amrouch, N. Du, A. Gebregiorgis, S. Hamdioui, and I. Polian, "Towards reliable in-memory computing: From emerging devices to post-von-neumann architectures," in *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 1–6, IEEE, 2021.

[44] P. Bowen, G. Regev, N. Regev, B. Pedroni, E. Hanson, and Y. Chen, "Analog, in-memory compute architectures for artificial intelligence," *arXiv preprint arXiv:2302.06417*, 2023.

[45] M. H. I. Chowdhuryy, M. R. H. Rashed, A. Awad, R. Ewetz, and F. Yao, "Ladder: Architecting content and location-aware writes for crossbar resistive memories," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 117–130, 2021.

[46] A. Mishchenko *et al.*, "Abc: A system for sequential synthesis and verification." "http://www.eecs.berkeley.edu/alanmi/abc".

[47] M. R. H. Rashed, S. K. Jha, and R. Ewetz, "Logic synthesis for digital in-memory computing," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2022.

[48] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, *et al.*, "Freepdk: An open-source variation-aware design kit," in *2007 IEEE international conference on Microelectronic Systems Education (MSE'07)*, pp. 173–174, IEEE, 2007.

[49] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[50] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[51] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, 2015.

[52] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 802–815, 2019.

[53] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *2017 IEEE international symposium on high performance computer architecture (HPCA)*, pp. 541–552, IEEE, 2017.