Execution Sequence Optimization for Processing In-Memory using Parallel Data Preparation

Muhammad Rashedul Haq Rashed*, Sven Thijssen*, Dominic Simon*, Sumit K. Jha†, Rickard Ewetz*
*Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL, USA
†Computer Science Department, Florida International University, Miami, FL, USA
{muhammad.rashed,sven.thijssen,dominic.simon}@ucf.edu,sumit.jha@fiu.edu,rickard.ewetz@ucf.edu

ABSTRACT

Processing in-memory (PIM) promises to unleash unprecedented computing capabilities for high-data-rate applications. Computation using PIM is performed by breaking down computationally expensive operations into in-memory kernels that can be efficiently executed using non-volatile memory. Logic styles such as MAGIC require that each output memory cell is prepared for evaluation before executing the functional logic operation. State-of-the-art synthesis algorithms perform the preparation immediately after memory cells have expired. Unfortunately, this results in that columns of cells are prepared greedily, instead of leveraging efficient parallel data preparation instructions. In this paper, we propose the PREP framework that maximizes the opportunities for parallel column preparation using execution sequence optimization. The key idea of the framework is to postpone data preparation instructions until there are no available prepared cells. Next, the accumulated memory cells are prepared in parallel to release the memory for functional evaluations. The framework is capable of exploring a frontier of area-performance solutions. The PREP framework is evaluated using 15 benchmarks from the SuiteSparse library. Compared with state-of-the-art synthesis tools, energy consumption and latency are respectively reduced by 27% and 25% with no additional cost in crossbar memory.

ACM Reference Format:
Muhammad Rashedul Haq Rashed, Sven Thijssen, Dominic Simon, Sumit K. Jha, and Rickard Ewetz. 2024. Execution Sequence Optimization for Processing In-Memory using Parallel Data Preparation. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657348

1 INTRODUCTION

The next wave of scientific discovery is predicated on scaling up scientific simulation capabilities. The success of AlphaFold 2.0 in predicting 3D protein structures comes from the convergence of new AI algorithms and scaled-up computing resources [4]. Scientific simulations for complex physical systems such as weather forecasting [1], drug discovery [14], and combustion [8] take days or months using today's high-performance computing systems.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3657348

Table 1: Evaluation with 32-bit Fixed-Point Multiplication.

Framework	Preparation WRITEs	Functional WRITEs	#Total WRITEs
LOGIC [15]	5549	10046	15595
PREP (**proposed**)	1271	10046	11317
Improvement	**77%**	**0%**	**27%**

With the near end of technology scaling, there is a surging interest in exploring emerging computing paradigms such as photonic [20], quantum [13], and in-memory computing [18].

Processing in-memory (PIM) can be performed using analog matrix-vector multiplication, digital Boolean functions, and content addressable memories (CAMs). While analog in-memory computing is energy-efficient, digital in-memory paradigms are more promising for scientific computing applications with strict requirements on computational accuracy. Logic styles for digital in-memory computing include OR-plane logic [16], PATH [22], bitwise-in-bulk [11], IMPLY [3], and MAGIC [10]. OR-plane logic and PATH-based computing are efficient for executing irregular Boolean functions [22]. On the other hand, bitwise-in-bulk, IMPLY, and MAGIC paradigms are ideal for accelerating matrix-vector multiplication operations due to their highly parallel nature [9, 11]. The MAGIC logic style is currently the most advantageous because it does not require expensive peripheral circuitry.

To accelerate expensive matrix-vector multiplication operations, the computation is first decomposed into in-memory compute kernels. Next, the in-memory compute kernels are ordered into an execution sequence. The MAGIC logic style supports INV, NOR2, and multi-input NOR operations. Each operation requires the preparation of an output device and the execution of a logic instruction. Both the preparation and the functional evaluation involve performing a WRITE operation. The state-of-the-art synthesis tools perform the cell preparation immediately when any memory cell has expired. However, this results in columns of cells being prepared greedily. At the same time, the cell preparation is performed using WRITE operations of type SET, which can be efficiently performed in parallel [23]. Hence, there exists an opportunity to improve performance by leveraging parallel SET operations for content preparation.

In this paper, we propose PREP, a framework that maximizes the use of parallel content preparation instructions using execution sequence optimization. The synthesis tool compiles the desired computation into a graph of in-memory operations with predecessor constraints. Sequence optimization is performed by postponing (or accumulating) cell preparation instructions while processing functional operations. Parallel cell preparation instructions are performed when no prepared cells are available for functional evaluations. The PREP framework explores the Pareto optimal frontier of area-performance solutions. Table 1 shows that the PREP framework can reduce the number of total in-memory operations for

32-bit fixed-point multiplication by 27% compared to the state-of-the-art [15]. The main innovations of the paper can be summarized, as follows:

- The observation that parallel cell preparation can be used to improve the performance of in-memory computing paradigms such as IMPLY and MAGIC.
- A graph formulation of the sequencing problem that allows both performance and memory utilization to be optimized using parallel cell preparation instructions.
- The experimental evaluation shows that the proposed paradigm speeds up arithmetic operations by 26% on average without any additional crossbar memory overhead.
- On 15 scientific computing applications, the PREP framework achieves a 25% speedup and 27% energy efficiency compared to the state-of-the-art paradigm.

The remainder of the paper is organized as follows: preliminaries and the motivation are presented in Section 2. The problem formulation is presented in Section 3. The PREP framework is introduced in Section 4. The experimental evaluation of the framework is discussed in Section 5. The paper is concluded in Section 6.

2 PRELIMINARIES

In this section, we first explain how MAGIC can be used to evaluate Boolean functions. Next, we showcase the opportunities for parallel data preparation instructions.

2.1 Digital In-Memory Computing using MAGIC

Digital in-memory computing relies on performing Boolean operations using non-volatile memory devices. The execution of a NOR ($\overline{x_1 + x_2}$) and INV ($\overline{x_3}$) operation using MAGIC is shown in Figure 1. The paradigm performs computation using an initialization step and an evaluation step. In the initialization step, the Boolean input operands are first programmed into a row of non-volatile memory (NVM) cells. Next, cells dedicated to storing the output of the logic operations are *prepared* by programming them to a low resistance state (LRS) using a SET operation. During the evaluation step, the NOR/INV operations are realized by applying controlled voltages to the input cells and grounding the output cells. MAGIC is an attractive logic style as it offers a high order of parallelism using SIMD. In a crossbar architecture, each row of the crossbar can execute independent NOR/INV netlists in parallel, which can accelerate applications dominated by matrix-vector multiplication operations.

Next, we explain how cell preparation instructions are used to reduce memory utilization when executing a sequence of MAGIC

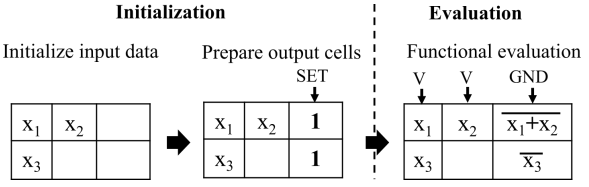

Figure 1: Steps for evaluating NOR/INV operations using MAGIC. Each box represents a memristor cell within crossbar.

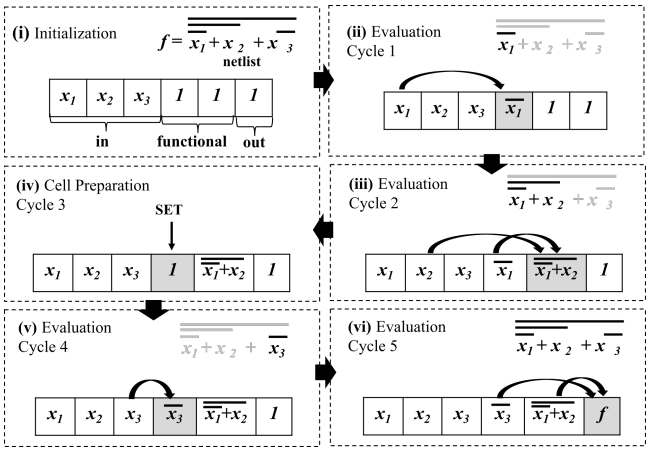

Figure 2: Execution sequence of a MAGIC netlist. (i) Crossbar row initialization and, (ii-vi) sequential execution of NOR/INV/SET operations. The SET operations allow memory cells to be reused after the content has expired.

operations, which is shown in Figure 2. The figure shows the execution of a NOR/INV netlist $f = \overline{\overline{x_1 + x_2} + \overline{x_3}}$. Figure 2(i) shows the initialization step of MAGIC where the input operands (x_{1-3}) are programmed into the input cells, and the functional and the output cells are initialized to 1 (LRS). The functional cells are used to evaluate the intermediate NOR/INV gates of the netlists. Figure 2(ii)–(vi) show the sequential evaluations of the NOR/INV gates and the intermediate cell preparation step. In the first two cycles, $\overline{x_1}$ and $\overline{\overline{x_1} + x_2}$ are evaluated, and the results are stored in the functional cells. After $\overline{\overline{x_1} + x_2}$ is evaluated, the result of $\overline{x_1}$ is no longer needed, and the memory cell storing the result can be freed and reused for future evaluations. If the memory cell was not reused, the memory requirements would have been increased by one. Therefore, in the third cycle, the functional memristor storing the $\overline{x_1}$ is prepared for reuse by using a SET operation which is shown in Figure 2(iv). In the next couple of cycles, the remaining NOR/INV operations are evaluated in Figure 2(v)–(vi). The example highlights that it is critical to reuse expired memory cells to minimize memory utilization.

2.2 Serial vs. Parallel Cell Preparation

The outlined approach of memory cell reuse (from previous work) aims to minimize the memory cell requirements. This means that as soon as the value stored in a functional cell expires i.e., no longer required for future executions, the algorithm immediately frees the cell and prepares it for reuse using a SET operation. Due to

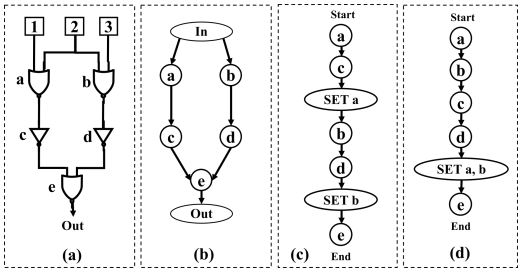

Figure 3: (a) An example NOR-INV netlist, (d) equivalent DAG of the netlist, (c) execution sequence with serial cell preparation, and, (d) execution sequence with parallel cell preparation.

the greedy nature of this cell preparation step, memory cells will almost exclusively be prepared using individual data preparation instructions. However, previous studies have shown that parallel SET operations are efficient and do not incur latency overhead [23]. We speculate that if some cell preparations are held off until later cycles, we can prepare cells using parallel SET operations, which in turn will reduce the total number of WRITE cycles. We illustrate the concept with an example in Figure 3.

Figure 3(a) shows an example MAGIC netlist with three primary inputs (1 – 3) and five NOR/INV gates (a–e). In Figure 3(b), an equivalent DAG representation of the netlist is shown. The inputs of the netlist are represented with a unified input source and the NOR/INV gates are represented with five nodes, a–e. The goal is to explore sequences for the node execution that yield a minimum number of cell preparation WRITEs. In Figure 3(c), we present an execution sequence with serial cell preparation operations. The sequence results in two SET operations to immediately prepare the expired cells. The sequence incurs 7 WRITE cycles in total. In Figure 3(d), we present an alternative execution sequence where we hold off immediate cell preparation to perform parallel cell preparation in a later cycle. The alternative sequence results in 6 WRITE cycles in total. This observation motivates us to develop a framework that maximizes the opportunity of parallel cell preparation to reduce the latency of the netlist execution.

3 PROBLEM FORMULATION

The objective of this paper is to leverage parallel data preparation instructions to improve the performance of digital in-memory computing using MAGIC. This is achieved by creating an additional optimization step in the synthesis flow for MAGIC, which is shown in Figure 4. The flow shows that an arithmetic operation is first compiled into an INV/NOR netlist [12]. Next, the execution sequence is optimized to minimize memory utilization. This is achieved by scheduling the functional operations such that memory cells are released as early as possible [15]. The output of this step is a netlist and the minimum required area (number of memory cells N). The proposed sequence optimization preserves the order of the functional operations while only optimizing the order of the data preparation instructions. The output is a netlist that contains parallel data preparation instructions. Next, the resulting execution sequence is utilized to perform processing in memory using MAGIC. The specific objectives of the proposed execution sequence optimization are, as follows:

(1) Given an execution sequence P and a cell count constraint N, optimize the execution sequence to minimize the total number of data preparation instructions. The number of instructions can be reduced by introducing parallel data preparation instructions.
(2) Determine the Pareto optimal frontier of execution sequences with optimal area performance.

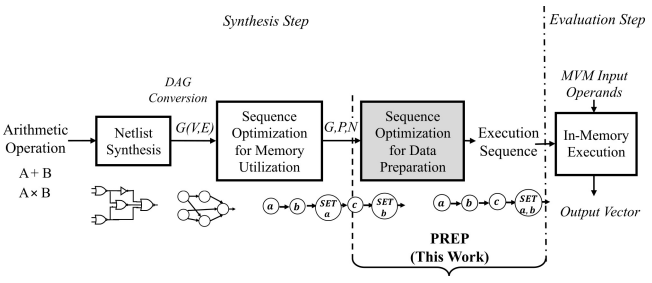

Figure 4: Overview of synthesis flow for MAGIC.

4 THE PREP FRAMEWORK

In this section, we introduce the PREP framework. The overview of the framework is shown in Figure 4. The main innovation of this paper is the observation that parallel data preparation instructions can be performed. No previous work has considered the aspect of parallel data preparation for in-memory computing. Our approach is based on the insight that data preparation instructions are not required to be performed as soon as a memory cell has expired. Instead, we delay performing cell preparation instruction until there are no memory cells available for functional evaluations. This results in that the opportunities for parallel data preparation are maximized. The proposed algorithm targets the use of the minimum memory cell count N. However, by considering relaxed constraints on the cell count, the Pareto optimal frontier of performance-area solutions can be explored.

In the following, we outline the details of the proposed sequence optimization in Section 4.1. Next, we illustrate the algorithm with an example in Section 4.2.

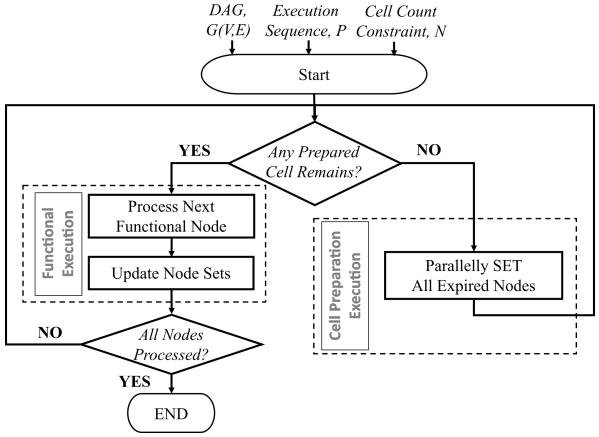

Figure 5: Flowchart of PREP framework.

4.1 Sequence Optimization for Data Preparation

In this section, we explain the proposed sequence optimization for data preparation. The input to the algorithm is an area-optimized execution sequence P and a cell count constraint N. The algorithm also requires the DAG $G = (V, E)$ to capture the predecessor constraints of the operations. By default, we use the execution sequence and the minimum cell count obtained from the state-of-the-art LOGIC framework [15]. The output is a new execution sequence that utilizes parallel data preparation instructions while still satisfying the cell count constraint. The flowchart for our proposed algorithm is shown in Figure 5.

The proposed algorithm maintains three sets of nodes, active nodes A (memory cells storing intermediate data), expired nodes X (cells with expired intermediate data), and cells prepared for functional evaluation. For simplicity, we simply show the count for

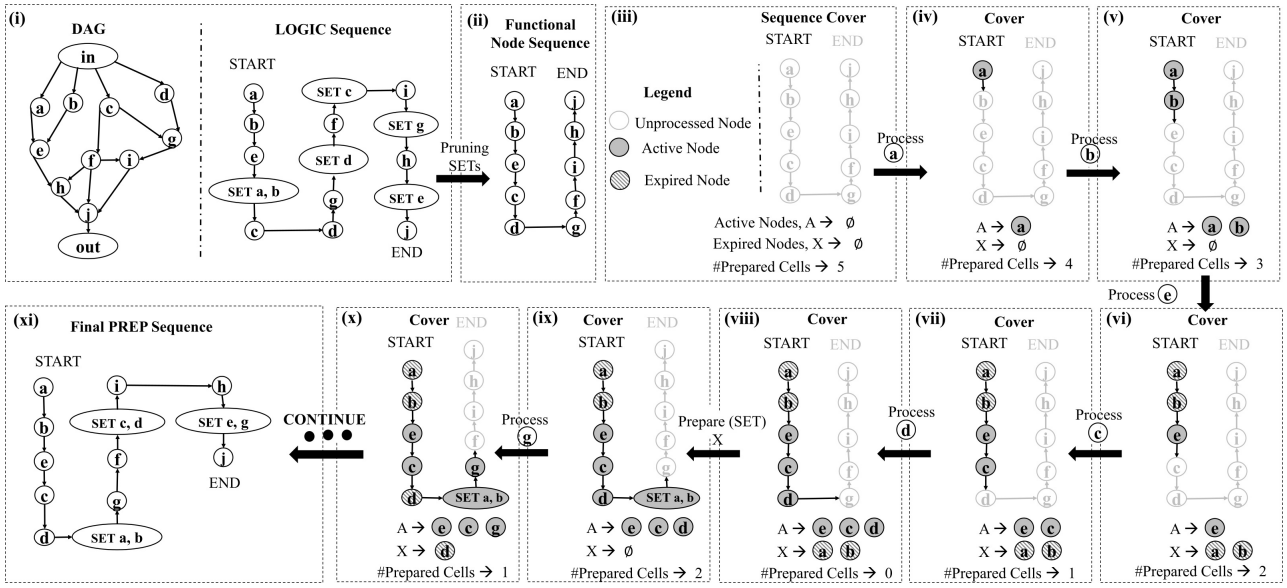

Figure 6: The workflow of the sequence optimization for the data preparation algorithm. (i) Target DAG and area-optimized LOGIC sequence [15], (ii) pruned sequence with only the functional nodes, (iii) - (viii) and (x) workflow of cell count constrained ($N = 5$) processing of functional nodes, (ix) parallel cell preparation of expired nodes and, (xi) optimized PREP sequence.

the prepared cells in the example in Section 4.2. We first discard all the data preparation instructions from the execution sequence. Next, we process functional operations one by one. If there exists a prepared memory cell, we schedule the next in-memory compute operation. Once the operation is scheduled, we update the active node and the expired node sets. The output of the considered operation is added to the active node set. Next, we use the DAG to check if any of the active nodes have expired, i.e., the intermediate result is not needed to compute any future output. Next, we move on to processing the next in-memory compute operation. Now we return to the case that there are no prepared memory cells available. Then, we prepare all the expired nodes and update the prepared cell count and the active and expired node sets. The same functional operation is considered in the next iteration of the algorithm.

4.2 Example of Algorithm

The workflow of the algorithm is illustrated with an example in Figure 6. Figure 6(i) shows a target DAG and the corresponding area-optimized execution sequence of the DAG nodes using the LOGIC framework [15]. The LOGIC framework adopts a greedy approach for sequence generation and releases expired memory as soon as possible. As a result, the generated sequence consists of five cell preparation cycles. While the parallel SET operations are not created intentionally, it is not uncommon that parallel resets with 2-3 cells are created due to that the nodes expire simultaneously. The sequence optimization algorithm aims to maximize the parallel cell preparation operations. First, the LOGIC sequence is pruned to remove all the SET instructions in Figure 6(ii). Next, the functional nodes within the pruned sequence are sequentially processed. For this example, we consider a cell count threshold of $N = 5$. In each step, the algorithm checks if any prepared cells are remaining. The algorithm also keeps track of the set of active nodes A and the set of expired nodes X. In Figure 6(iii)–(viii), we sequentially process the first five functional nodes a→b→e→c→d of the sequence. The figures show the changes in sets A and X and the available prepared cell count for each node processing. After the d is processed in Figure 6(viii), the count of the prepared cell is 0. Therefore cell preparation is required. In Figure 6(ix), all expired nodes in X (a and b) are parallelly SET to prepare memory cells. This sequential node processing and memory preparation steps are continued until all the functional nodes are processed in the order dictated by the pruned LOGIC sequence. The final sequence for the PREP framework is shown in Figure 6(xi). The figure shows that the number of cell preparation cycles is reduced by two cycles in the optimized sequence. This improvement directly translates to latency and energy improvements within the PREP framework.

5 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the performance of the PREP framework. We conduct the experiments on an octa-core machine with 3.60 GHz Intel Core i9 processor with NVIDIA RTX 2070 and 64 GB RAM. We define the arithmetic operations using Verilog scripts. We utilize the ABC tool [12] with a custom NOR/INV library to generate the MAGIC netlist. We develop the sequence optimization tool using C++.

We compare the performance of the PREP framework with respect to the state-of-the-art in-memory computing framework LOGIC [15]. We also utilize the latency-optimized frameworks in Talati et al. [21] (for addition) and Haj-Ali et al. [6] (for multiplication) for comparative evaluation. Note that the works in [6, 21] are centered on sub-optimal manual decomposition of computation. For a fair evaluation, we adopt the automated synthesis approach and a similar architecture to the state-of-the-art LOGIC paradigm. This also enables us to perform a direct comparison among the competing frameworks. The power-area costs of micro-architectural components are appropriately adapted from [9, 19]. Cross-architectural data transfer cost is estimated using the CACTI 7 tool [2].

In the subsequent sections, we first perform a sensitivity analysis to determine the best parameter values for the PREP framework.

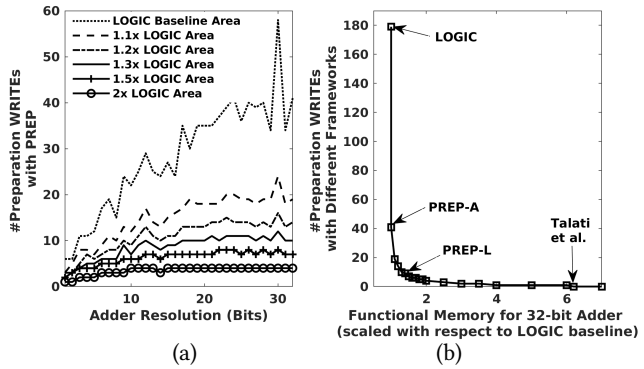

Figure 7: (a) Trend of #preparation WRITEs using the PREP framework for variable area (#cells) threshold and, (b) #preparation WRITEs vs area threshold for different frameworks.

Next, we evaluate the performance of the PREP framework for MVM operations using 15 matrices from the SuiteSparse benchmark suit [5].

5.1 Sensitivity Analysis

In this section, we first perform a sensitivity analysis by varying the area (#cells) threshold within the PREP framework. The goal of this experiment is to observe the impact of the parameter changes within the PREP framework. Next, we perform an experiment to quantify the degree of parallel cell preparation operations within different frameworks.

5.1.1 Sensitivity Analysis for Cell Count Constraint. There exists a direct correlation between the area (cell count) threshold and the required number of SET operations within the PREP framework. This is intuitive because, for a higher area threshold, the PREP framework can execute more functional operations before a cell preparation step is warranted. Figure 7(a) shows the trend of preparation WRITE operations for different adder resolutions and different area thresholds. It can be observed from the figure that for higher area thresholds, the number of preparation WRITEs significantly reduces. For instance, a mere 10% increase in area compared to the LOGIC baseline area can reduce the number of preparation WRITEs by 45% on average. Note that the graphs contain some spikes for some adder resolutions (e.g., for the 30-bit adder). This is due to that the execution sequencing algorithm within the LOGIC framework is a greedy algorithm with a randomized component. This leads to that for some adders the number of cell preparation WRITEs spikes while attempting to optimize the area reuse.

An attractive feature of PREP is that the performance of the framework can be tailored by tuning the area threshold. The full potential of the PREP framework is illustrated in Figure 7(b) where the change in the required number of cell preparation WRITEs for 32-bit addition using different memory allocations is illustrated. The Figure highlights the LOGIC framework where the area is minimum while the number of cell preparations is the highest. On the other hand, the latency-optimized algorithm in Talati et al. [21] incurs the minimum number of cell preparations while incurring $6.2x$ more area compared to the LOGIC framework. The PREP framework introduces an opportunity to customize area-latency performance. The figure highlights two points *PREP-A* and *PREP-L*. *PREP-A* incurs the same area as the LOGIC framework while

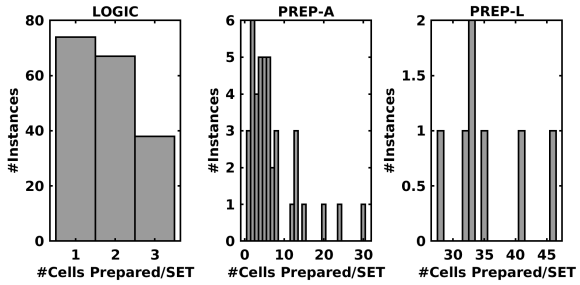

Figure 8: Parallel cell preparations for 32-bit addition.

reducing the number of cell preparation WRITEs using the PREP data preparation algorithm. *PREP-L* further reduces the number of cell preparation WRITEs while incurring $1.5x$ area compared to the LOGIC framework. We do not select a higher area threshold value for *PREP-L* because the improvement in latency tends to saturate as shown in the figure. We use *PREP-A* and *PREP-L* for MVM evaluation in the next section.

5.1.2 Parallel Cell Preparations. Figure 8 shows how many cells are parallelly prepared in each data preparation instruction within different frameworks. The figure shows that for 32-bit adder operation, the LOGIC framework performs in total of 179 cell preparation operations. All of these cell preparation steps are restricted to a parallel cell preparation of $1-3$ cells. On the other hand, the *PREP-A* framework incurs 41 cell preparation cycles and parallelly prepares up to 30 cells. Finally, the *PREP-L* incurs only 7 cell preparation cycles and parallelly sets cells in the range between $28-46$ in each instance.

5.2 Evaluation with MVM Applications

In this section, we evaluate the performance of the PREP framework for MVM operations. In-memory computing is a promising solution to accelerate data-intensive applications that are dominated by MVM operations. For instance, the systems of linear equations can be solved using iterative refinement algorithms such as the conjugate gradient method (CG) [7] and generalized minimal residual method (GMRES) [17]. In each iteration of these methods, an expensive MVM operation is performed. In-memory computing platforms can greatly accelerate this MVM process by performing parallel in-situ operations.

To perform this comparative evaluation, we select 15 MVM-based applications from the SuiteSpare benchmark suits [5]. The applications are listed in Table 2. The applications are selected from a wide range of scientific domains. The matrices within the

Table 2: Overview of benchmarks from the SuiteSparse Matrix Collection [5].

Applications	Systems	Matrix Dimensions	#Non-zeros
eris1176	Power Network Problem	1176 × 1176	18552
cegb2919	Structural Problem	2919 × 2919	321543
raefsky1	Computational Fluid Dynamics	3242 × 3242	293409
fxm3_6	Optimization Problem	5026 × 5026	94026
Na5	Theoretical/Quantum Chemistry	5832 × 5832	305630
EX5	Combinatorial Problem	6545 × 6545	295680
fp	Electromagnetics Problem	7548 × 7548	834222
ex40	Computational Fluid Dynamics	7740 × 7740	456188
benzene	Theoretical/Quantum Chemistry	8219 × 8219	242669
bcsstk33	Structural Problem	8738 × 8738	591904
graham1	Computational Fluid Dynamics	9035 × 9035	335472
net25	Optimization Problem	9520 × 9520	401200
bundle1	Computer Graphics/Vision	10581 × 10581	770811
Si10H16	Theoretical/Quantum Chemistry	17077 × 17077	875923
Goodwin_040	Computational Fluid Dynamics	17922 × 17922	561677

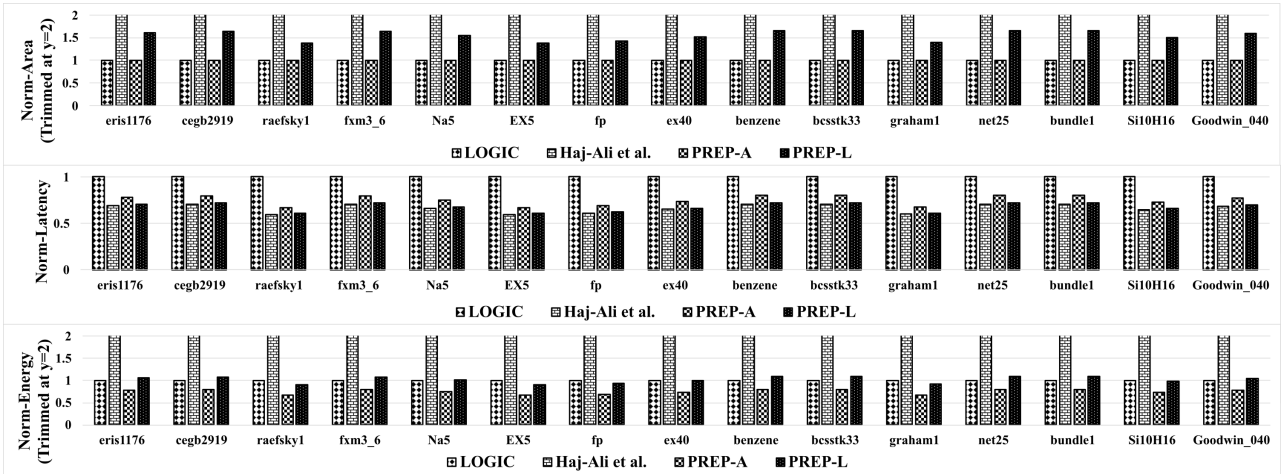

Figure 9: Area-latency-energy overhead evaluation of different frameworks.

applications also have different data patterns, which impact the performance of the computing paradigms.

The comparative area-latency-energy performance of different paradigms is presented in Figure 9. The y-axis in the figures for area-energy evaluation is trimmed at $y = 2$ for illustration purposes. Our experiments show that the Haj-Ali et al. [6] framework requires 12X more area on average compared to the LOGIC framework. This is expected as the Haj-Ali et al. framework adopts a latency-optimized algorithm where the area is unconstrained. On the other hand, due to the cell count constraint on the PREP-A framework, the area cost is equal to that of the LOGIC framework. The figure also shows that the latency performance of the LOGIC framework is the worst of the three paradigms due to its greedy cell preparation operations. However, the Haj-Ali et al. framework and PREP-A and PREP-L frameworks show comparable latency performances. This is due to that the number of cell-preparation WRITEs is greatly reduced within the PREP framework. With the data preparation optimization algorithm, the PREP-A and PREP-L frameworks achieve 25% and 32% speedup compared to the LOGIC framework, respectively. This speed-up also translates into a 27% improvement in energy consumption within the PREP-A framework, compared to the LOGIC framework. Conversely, due to the massive hardware overhead, the Haj-Ali et al. framework is the least energy-efficient of the three frameworks. This significant improvement in energy consumption within the PREP-A framework is the result of its faster processing capability, which minimizes the accumulated energy consumption of the architectural components.

6 CONCLUSION

In this work, we develop an area-constrained latency-optimized in-memory computing framework to accelerate arithmetic operations. The framework is centered on performing parallel cell preparation operations on the expired computing nodes which is significantly more efficient than serial cell preparation operations. The framework offers superior energy and latency performance compared to the state-of-the-art paradigm while incurring no additional crossbar memory overhead. Additionally, the performance of the framework can be customized by tuning the constraints. In future work, we plan to implement the framework to accelerate new applications.

ACKNOWLEDGMENTS

This work was in part supported by NSF awards #2319399, #2408925, and #2404036.

REFERENCES

[1] V. Balaji. Climbing down charney's ladder: machine learning and the post-dennard era of computational climate science. *Philosophical Transactions of the Royal Society A*, 379(2194):20200085, 2021.
[2] R. Balasubramonian et al. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM TACO*, 14(2):1–25, 2017.
[3] J. Borghetti et al. 'memristive' switches enable 'stateful' logic operations via material implication. *Nature*, 464(7290):873–876, 2010.
[4] E. Callaway. What's next for the ai protein-folding revolution. *Nature*, 604:234–238, 2022.
[5] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.
[6] A. Haj-Ali et al. Efficient algorithms for in-memory fixed point multiplication using magic. In *2018 IEEE ISCAS*, pages 1–5. IEEE, 2018.
[7] M. R. Hestenes. The conjugate gradient method for solving linear systems. In *Proc. Symp. Appl. Math VI, American Mathematical Society*, pages 83–102, 1956.
[8] M. Ihme et al. Combustion machine learning: Principles, progress and prospects. *Progress in Energy and Combustion Science*, 91:101010, 2022.
[9] M. Imani et al. Floatpim: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th ISCA*, pages 802–815, 2019.
[10] S. Kvatinsky et al. Magic—memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(11):895–899, 2014.
[11] S. Li et al. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *DAC*, pages 1–6. IEEE, 2016.
[12] A. Mishchenko et al. Abc: A system for sequential synthesis and verification. "http://www.eecs.berkeley.edu/alanmi/abc".
[13] E. National Academies of Sciences, Medicine, et al. Quantum computing: progress and prospects. 2019.
[14] M. Pandey et al. The transformational role of gpu computing and deep learning in drug discovery. *Nature Machine Intelligence*, 4(3):211–221, 2022.
[15] M. R. H. Rashed et al. Logic synthesis for digital in-memory computing. In *Proceedings of the 41st IEEE/ACM ICCAD*, pages 1–9, 2022.
[16] M. R. H. Rashed et al. Stream: Towards read-based in-memory computing for streaming based processing for data-intensive applications. *IEEE TCAD*, 2023.
[17] Y. Saad and M. H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.
[18] A. Sebastian et al. Memory devices and applications for in-memory computing. *Nature nanotechnology*, 15(7):529–544, 2020.
[19] A. Shafiee et al. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *2016 ACM/IEEE 43rd ISCA*, pages 14–26, 2016.
[20] B. J. Shastri et al. Photonics for artificial intelligence and neuromorphic computing. *Nature Photonics*, 15(2):102–114, 2021.
[21] N. Talati et al. Logic design within memristive memories using memristor-aided logic (magic). *IEEE TNANO*, 15(4):635–650, 2016.
[22] S. Thijssen et al. Path: Evaluation of boolean logic using path-based in-memory computing. In *Proceedings of the 59th ACM/IEEE DAC*, pages 1129–1134, 2022.
[23] C. Xu et al. Overcoming the challenges of crossbar resistive memory architectures. In *2015 IEEE 21st HPCA*, pages 476–488. IEEE, 2015.