Architectural Study, Benchmarking and Comparative Analysis of In-Memory Computing (IMC) Simulators

A Comprehensive Study of Open Source IMC Simulators' Architectures and Their Benchmark Performance

Vraj Shah
22110292
Computer Science Eng.

Shubham Agrawal
22110249
Computer Science Eng.

Dewansh Singh Chandel
22110072
Computer Science Eng.

Sawale Sumeet Shivaji
22110234
Computer Science Eng.

Abstract—**In-Memory Computing (IMC) is a transformative paradigm that enhances data-intensive applications by integrating memory and computation, thus addressing the limitations of traditional Von Neumann architectures. This project investigates both the architectural foundations and the performance of selected IMC simulators, providing a comprehensive comparative analysis. By studying the unique design principles underlying each simulator's IMC architecture, we examine how factors such as memory layout, processing capabilities, and integration techniques impact computational efficiency. Furthermore, we benchmark each simulator against metrics like latency, energy consumption, and throughput to highlight trade-offs and performance variations. Our study offers a dual perspective—combining architectural insights with empirical benchmarks—that serves as a guide for researchers and engineers in selecting and deploying IMC simulators for optimized, application-specific solutions.**

I. Introduction

In-Memory Computing (IMC) represents a significant shift from traditional Von Neumann architectures by merging data storage and computational tasks within the same memory location. This integration reduces data movement, which is a primary bottleneck in data-intensive applications, and enables significant improvements in latency, energy efficiency, and overall computational throughput. IMC has gained considerable traction in fields such as artificial intelligence (AI), machine learning (ML), and large-scale data analytics, where the demand for real-time processing and efficient energy usage is paramount.

Simulators play an essential role in advancing IMC by providing researchers and engineers with platforms to model, test, and analyze the effectiveness of various IMC architectures before hardware implementation. Open-source IMC simulators, in particular, offer accessible and modifiable environments for exploring IMC principles, examining architectural trade-offs, and evaluating performance metrics under diverse configurations. However, the lack of standardized frameworks for assessing IMC simulators complicates the selection of the most suitable simulator for specific applications.

This paper aims to address this gap by conducting a comprehensive architectural study and benchmarking analysis of several widely used open-source IMC simulators. By comparing these simulators based on architecture, memory layout, processing capabilities, and integration techniques, we provide insights into their efficiency and suitability for specific applications. Furthermore, our benchmarking results focus on critical metrics, including latency, energy consumption, and throughput, to offer a well-rounded evaluation of each simulator's performance. This study serves as a practical guide for researchers and engineers in selecting appropriate IMC simulators for optimized, application-specific deployments.

II. Objectives Planned

A. Simulator Selection

B. Definition of Benchmarking Programs

C. Definition of Performance Metrics

D. Performance Benchmarking

E. Comparative Analysis and Insights for Practical Applications

III. Objectives Achieved

A. Study and Understand In-Memory Computing (IMC)

- Gained a deep understanding of In-Memory Computing (IMC), including its fundamental principles, architectural advantages, and the motivations behind its development, to effectively contextualize and inform the subsequent architectural study and benchmarking of IMC simulators.

B. Simulator Selection

- After evaluating various options, successfully identified and selected several open-source simulators for benchmarking, ensuring they meet our criteria in terms of functionality, community support, and relevance to IMC architectures

C. *Comprehensive Study of IMC Architectures*

- Conducted an in-depth study of the architecture of 3-4 selected IMC simulators, focusing on their unique memory layouts, data processing techniques, and design trade-offs.

D. *Definition of Benchmarking Programs*

- Custom-developed programs to create a diverse set of benchmarking tests that target both general and IMC-specific workloads.

E. *Definition of Performance Metrics*

- Defined key performance metrics, including execution time, memory usage, and scalability, that will allow us to comprehensively evaluate and compare the performance of the simulators, with a focus on the specific strengths of IMC.

F. *Performance Benchmarking*

- Conducted benchmarking tests across all simulators using the standardized set of tasks and evaluated them based on the performace metrics.

G. *Comparative Analysis and Insights for Practical Applications*

- Provided a comparative analysis of the simulators, offering practical recommendations on selecting IMC tools for specific applications based on both architecture and performance characteristics.

IV. In Memory Computing

A. *Principle*

In-Memory Computing (IMC) is an emerging computational paradigm that addresses the ”memory wall” problem—the data transfer bottleneck that arises in traditional Von Neumann architectures. In these conventional systems, data is constantly moved back and forth between the processor and memory, leading to significant latency and power consumption. IMC seeks to overcome these limitations by integrating processing capabilities directly within the memory elements, thereby minimizing data movement and improving overall efficiency.

By co-locating memory and computation, IMC enables faster processing, particularly for applications requiring high data throughput, such as artificial intelligence (AI), machine learning (ML), and big data analytics. Instead of transferring data to a central processing unit (CPU) for computation, data operations occur within the memory cells themselves, drastically reducing delays and energy costs associated with data movement.

B. *Key Concepts*

- **Energy Efficiency** - By reducing data transfers between memory and processor, IMC significantly cuts down on energy consumption. This is particularly advantageous for applications running on battery-powered devices or systems where power efficiency is crucial.
- **Reduced Latency** - Processing data directly within memory minimizes latency, leading to faster computational speeds. For real-time applications, such as edge computing and autonomous vehicles, this low-latency characteristic is invaluable.
- **Scalability for Data-Intensive Applications** - IMC is well-suited to data-heavy tasks, enabling the scalable and efficient processing required by big data and machine learning algorithms.
- **Parallelism** - With the inherent parallel nature of memory cells, IMC architectures support massive parallel processing, improving the efficiency of complex operations like matrix multiplications used in neural networks.

C. *Application*

IMC has gained significant traction across various domains due to its efficiency in handling large datasets and computational tasks while minimizing energy consumption and latency. Some of the prominent applications include:

1. Artificial Intelligence (AI) and Machine Learning (ML): IMC is particularly advantageous for AI and ML applications, which often involve processing massive amounts of data through complex computations. Key reasons IMC is valuable in this field include:

- **Efficient Matrix Operations:** AI and ML rely heavily on matrix and vector multiplications, especially in neural networks and deep learning models. IMC can handle these operations within memory arrays, accelerating computations and improving efficiency.
- **Reduced Data Bottleneck:** Training large models typically involves substantial data movement between the CPU and memory, resulting in latency and energy costs. IMC minimizes these bottlenecks by performing computations directly in memory, making it more energy-efficient and faster.
- **Scalability for Large Models:** With increasing model complexity, IMC’s scalability is beneficial, as it can process data in parallel and accommodate the growing demands of larger AI models.

IMC has shown significant promise in accelerating tasks such as image recognition, natural language processing, and real-time data processing, where speed and efficiency are critical.

2. High-Performance Computing (HPC): In high-performance computing (HPC), speed and data throughput are crucial, especially in fields like scientific computing, financial modeling, and weather simulation. IMC is well-suited for HPC due to:

- **High Throughput:** HPC applications often involve tasks that require handling extensive datasets with high-speed computations. IMC architectures support this by enabling parallel processing within memory arrays, which increases throughput and decreases computation time.
- **Energy Efficiency for Large Data Sets:** HPC workloads are typically energy-intensive. By reducing data transfers and performing calculations in-memory, IMC offers an energy-efficient solution for HPC applications, making it cost-effective and sustainable.
- **Improved Latency for Real-Time Analysis:** HPC tasks such as simulations and data analysis often require quick data access and minimal latency. IMC's ability to perform computations close to where data is stored minimizes latency, providing faster results in real-time applications.

This application of IMC is particularly valuable in environments where rapid data processing is essential, such as financial trading systems, molecular dynamics simulations, and climate modeling.

3. Edge Computing: Edge computing involves processing data closer to the data source, typically in IoT devices or decentralized locations, to reduce latency and minimize data transfer to centralized data centers. IMC plays a crucial role here for the following reasons:

- **Low Power Consumption:** Edge devices often run on limited power sources, such as batteries. IMC's energy efficiency is ideal for edge computing, allowing devices to operate for extended periods without frequent recharging or energy replenishment.
- **Real-Time Data Processing:** Many edge applications, such as autonomous vehicles, security systems, and health monitoring devices, require immediate data processing. IMC's low-latency architecture enables real-time computation at the edge, which is critical for responsiveness and reliability.
- **Reduced Data Transmission:** Edge computing benefits from reducing the need to send data to a central server for processing. By handling computations locally with IMC, edge devices can perform complex tasks on-site, minimizing bandwidth usage and enhancing data privacy.

IMC's advantages align well with the needs of edge computing, making it a powerful solution for IoT devices, smart sensors, wearable technology, and any scenario where fast, localized data processing is required.

D. Challenges

- **Hardware Limitations** - Incorporating logic functions into memory cells is technically complex and may result in trade-offs in memory density and reliability. The design of robust IMC hardware remains a challenging aspect of scaling IMC technology.
- **Programming Complexity** - IMC architectures may require new programming models and compilers to efficiently map algorithms onto the IMC hardware, adding complexity to software development.
- **Thermal Management** - Higher power densities in IMC systems can lead to increased heat generation, posing challenges for thermal management and reliability.
- **Limited Arithmetic Precision** - Due to the constraints of in-memory devices, IMC often relies on approximate or low-precision computations, which may not be suitable for all applications, especially those requiring high accuracy.

V. Types of IMC Architectures

A. SRAM-based IMC

In-memory computing (IMC) using SRAM (Static Random-Access Memory) architecture is an advanced computational technique aimed at reducing data transfer bottlenecks between the memory and processing units. This approach leverages SRAM as both a memory storage and a computational unit, enabling data processing directly within the memory array.

1) SRAM Architecture Basics: SRAM is a type of volatile memory that stores bits in a bistable latching circuit, typically composed of six transistors (6T SRAM cell). This structure allows SRAM to be:

- **Fast**: SRAM can access data very quickly compared to other types of memory, like DRAM.
- **Power-efficient**: SRAM doesn't need to be periodically refreshed as DRAM does, leading to power savings.
- **Stable**: Data is retained in each SRAM cell as long as power is supplied.

The structure of an SRAM cell makes it inherently suitable for bitwise operations due to its reliance on transistors, which can act as binary switches.

2) Functioning of SRAM-based IMC: In an IMC system, the SRAM architecture is modified to support computational operations alongside storage.

- **Modified Periphery Circuits**: The periphery circuits in SRAM arrays are modified to include computation-enabling circuits, like sense amplifiers and summing nodes, which facilitate arithmetic operations directly within the memory array.
- **Row/Column Activation**: Computations are typically triggered by activating specific rows or columns within

the SRAM array. For example, activating multiple rows simultaneously can yield the result of an addition or a bitwise operation.

- **Data Encoding**: Data is encoded and stored in a way that aligns with the intended operations. For instance, binary-encoded data is ideal for bitwise operations, while analog encoding might be employed for approximate computing tasks.
- **Sense Amplifiers and Adders**: Sense amplifiers, originally designed for reading data from memory cells, can be adapted for comparison operations. Additional circuitry, such as adders and multipliers, can also be integrated at a minimal area and energy cost to execute specific mathematical operations.

3) SRAM for In-Memory Computing (IMC): In traditional computing, data is moved from the memory to the CPU or a dedicated ALU (Arithmetic Logic Unit) for processing, then back to memory. This movement creates latency and consumes energy. In SRAM-based IMC, data processing happens directly within the memory array, so SRAM performs both storage and computational roles.

- Bitwise Operations (AND, OR, XOR)
  - **Basic Mechanism:** SRAM cells store data in the form of binary values, which can be manipulated directly within the memory array using transistor logic to perform bitwise operations.
  - **Row/Column Activation:** To perform a bitwise operation between two values in SRAM (say two columns), both rows that contain the values can be activated simultaneously. When these rows are read, sense amplifiers connected to the bitlines detect combined voltage levels, allowing operations like AND, OR, or XOR.
  - **Sense Amplifiers:** The sense amplifiers (used to read the values) are configured to detect specific patterns in the bitline voltage, where they effectively perform the logical operation.
    - **AND Operation:** To compute A AND B, the array reads both rows, and a low voltage output signifies that both bits are 1.

      $$\text{Output} = A \wedge B$$
    - **OR Operation:** When reading the combined voltage across both rows, the amplifier is set to produce a high output if any of the bits are high.

      $$\text{Output} = A \vee B$$
    - **XOR Operation:** This can be done similarly by configuring sense amplifiers to detect exclusive high values on either bitline.

      $$\text{Output} = A \oplus B$$
- Matrix-Vector Multiplication
  - **Data Encoding:** Matrix and vector data are stored in SRAM cells across rows and columns.
  - **Parallel Row Activation:** During matrix-vector multiplication, several rows in the SRAM array are activated in parallel to perform multiple dot products simultaneously. Each row corresponds to a row in the matrix, and the vector is broadcast across columns.
  - **Analog Summation:** In analog IMC, the SRAM cells' read currents for a row are combined at the bitlines. This summation can approximate the multiplication and addition of values in the matrix row with the vector.

    $$\text{Dot Product} = \sum_i A_i \times B_i$$
  - **Digital or Analog Output:** The resulting values on the bitlines can be converted into digital signals (if more precise output is required) or processed in analog form, which is often faster for certain applications like neural networks.
- Analog Operations
  - **Analog Encoding:** In analog IMC, data can be encoded as varying voltages rather than binary 0s and 1s. This encoding allows SRAM cells to approximate certain calculations by manipulating electrical currents and voltages.
  - **Current Summation:** When multiple rows are activated, the currents in their respective bitlines sum up. For certain analog operations, this summed current represents the result of the computation (for example, an accumulation operation in neural networks).

    $$\text{Result} = \sum_i I_i$$
  - **Approximations and Noise Tolerance:** Because analog computation can introduce noise, it's typically used in applications where exact precision isn't required, such as machine learning. SRAM cells in analog mode take advantage of transistor properties, such as sub-threshold leakage currents, to represent and manipulate approximate values.

B. DRAM-based IMC

DRAM-based in-memory computing (IMC) has become a promising approach for high-performance, data-intensive applications. Unlike SRAM-based IMC, DRAM (Dynamic Random-Access Memory) is designed to store large amounts of data at a lower cost per bit, making it especially advantageous for applications requiring high-density memory.

1) DRAM Architecture Basics: DRAM consists of arrays of memory cells, where each cell is typically made up of a capacitor and a transistor. This capacitor-transistor setup stores bits as charge (1 or 0) on the capacitor. However, the charge on the capacitor decays over time, requiring periodic refreshing to retain data.

The core structure of DRAM includes:

- Memory Cells: Organized in a matrix with rows and columns.
- Wordlines and Bitlines: Rows are activated by wordlines, and columns are accessed by bitlines.
- Sense Amplifiers: Sense amplifiers detect the charge difference on the bitlines, allowing data to be read and written.

2) Functioning of DRAM-based IMC: For DRAM-based in-memory computing (IMC), the architecture also undergoes modifications to support computations alongside data storage.

- **Modified Periphery Circuits**: In DRAM-based in-memory computing (IMC), traditional periphery circuits are enhanced to handle computation by configuring sense amplifiers to detect logical states for operations and adding charge-sharing mechanisms to interpret shared charge levels for computational tasks.
- **Row/Column Activation**: For IMC, single-row activation is used for standard read/write operations, while dual or multi-row activation enables parallel bitwise operations like AND, OR, and summation by activating multiple rows simultaneously, facilitating computations within the memory array.
- **Data Encoding**: DRAM utilizes digital encoding (binary data storage) for exact bitwise operations and introduces analog encoding for approximate computations, storing various charge levels that enable a range of values, useful in applications like machine learning where precise accuracy is less critical.
- **Charge Accumulation and Sense Amplifiers**: In analog summation tasks like matrix-vector multiplications, multiple rows are activated to accumulate charge, which sense amplifiers detect as a summed value; this allows for bitwise operations such as AND, OR, and XOR by interpreting the cumulative charge levels from the activated rows.

3) DRAM for In-Memory Computing (IMC): Traditional DRAM architectures only support data storage, with computation handled by external processors. In DRAM-based IMC, however, the goal is to perform computation directly within the memory array, reducing data movement and improving processing speed. This is achieved by adapting DRAM's sense amplifiers and exploiting charge-sharing mechanisms for computation.

- Bitwise Operations (AND, OR, XOR)
  - **Row Activation and Dual-Row Operation:** Bitwise operations in DRAM leverage the simultaneous activation of two or more rows. By activating multiple rows, the charges on the corresponding cells interfere with each other, leading to logical bitwise outputs.
  - **Using Sense Amplifiers:** For example, if two rows are activated, the sense amplifier can interpret the resultant charge state on the bitline to perform an AND, OR, or XOR operation between the bit values.
  - **Example Process:**
    * **AND Operation:** If both cells in two rows hold a charge (representing binary '1'), the combined charge on the bitline will be higher, resulting in a '1' after amplification.
    * **OR Operation:** When either of the two cells holds a charge, the sense amplifier interprets this as a binary '1' on the output.
  - **Parallelism:** DRAM bitwise operations can be done in parallel across multiple rows and columns, increasing processing speed for data-intensive tasks like image processing and large-scale data search.
- Matrix-Vector Multiplication
  - **Data Placement:** Matrices are mapped across the DRAM cells, with rows and columns representing elements of the matrix.
  - **Row Activation and Charge Sharing:** In matrix multiplication, multiple rows in the DRAM are simultaneously activated. This action causes charge-sharing along the bitlines, allowing partial summations to take place directly within the memory array.
  - **Accumulation via Sense Amplifiers:** The sense amplifiers connected to each bitline detect and amplify the cumulative charge, effectively summing up the partial products across the rows. This behavior can be exploited for inner-product calculations, an essential part of matrix-vector multiplication.
  - **Analog Accumulation:** In some advanced designs, the DRAM cells are tuned to hold analog values (as varied charges on the capacitors). Activating multiple rows then results in an analog summation, where the combined charge represents the accumulated value, and the sense amplifiers can read this analog result.
- Analog Operations
  - **Analog Data Representation:** Analog IMC in DRAM stores data as varying charge levels on each capacitor, rather than just full charges for binary 1s and 0s. This configuration allows a range of values, which can represent continuous or approximate data.
  - **Charge Accumulation for Approximate Computation:** By activating multiple rows, the charges from several cells combine on the bitline, resulting in an analog summation. This technique is particularly useful for neural networks, where exact precision isn't always necessary.
  - **Tolerating Noise:** Analog operations in DRAM may introduce noise, but they are effective for applications

that can handle approximation. DRAM's large capacity allows for simultaneous analog summation across many cells, useful for processing high-dimensional data typical in AI models.

C. ReRAM-based IMC

ReRAM (Resistive Random Access Memory)-based in-memory computing (IMC) leverages the unique properties of resistive memory to perform computations directly in the memory array. Unlike DRAM and SRAM, which store data using capacitive or transistor-based mechanisms, ReRAM stores data as resistive states within metal-oxide layers, making it naturally suited for analog and digital computation.

1) ReRAM Architecture Basics: ReRAM (Resistive Random Access Memory) is a type of non-volatile memory, meaning it retains data even when power is turned off. Unlike traditional memory types, ReRAM stores information by changing the resistance of a metal-oxide material, which can switch between high-resistance and low-resistance states. This resistive state represents binary data (0 or 1), and in advanced cases, ReRAM cells can support multiple resistive levels to store more than one bit per cell.
Key ReRAM characteristics:

- **Crossbar Array Structure:** ReRAM cells are typically arranged in a crossbar array where each cell sits at the intersection of a wordline (horizontal wire) and a bitline (vertical wire).
- **Non-Volatile:** It retains data without needing a constant power supply.
- **Multi-Level Cell Capability:** Each cell can hold multiple resistance levels, useful for analog and approximate computing.
- **Low Power Consumption:** Due to its resistive nature, ReRAM consumes less power compared to other memory types for both storage and computation.

2) Functioning of ReRAM-based IMC: In an IMC setup, the ReRAM architecture is adapted to perform computational operations directly in memory, leveraging its unique resistive characteristics for both storage and processing.

- **Modified Periphery Circuits**: In ReRAM-based IMC, sense amplifiers interpret cell resistance levels for both binary (digital) and varied (analog) data; computations like summation and multiplication rely on Ohm's and Kirchhoff's laws, using applied voltages and measured currents.
- **Row/Column Activation**: Parallel activation of multiple ReRAM rows enables matrix multiplication and summation by reading cumulative currents on the bitlines; analog encoding of resistance levels supports approximate operations for tasks like machine learning.
- **Data Encoding**: ReRAM uses binary encoding (high and low resistance states) for precise bitwise operations, while analog encoding with varied resistance levels allows for multi-level or approximate computations in analog tasks.
- **Current Summation**: In analog operations like matrix-vector multiplication, ReRAM performs current summing on bitlines to represent weighted sums; voltage-controlled wordlines enable bitwise logic (AND, OR, XOR), which sense amplifiers interpret based on cumulative current levels.

3) ReRAM for In-Memory Computing (IMC):

- Bitwise Operations (AND, OR, XOR)
  - **Voltage Manipulation for Logic Operations:** Bitwise operations like AND, OR, and XOR are achieved by controlling the voltage levels across the memristors in the ReRAM array.
  - **Simultaneous Row Activation:** Similar to DRAM and SRAM, activating multiple wordlines allows ReRAM cells to combine their resistive states. The resultant voltage levels across the bitlines can then be interpreted as logical operations using sense amplifiers.
  - **Example Process:**
    - **AND Operation:** If both cells in the activated rows are in a low-resistance state (1), the combined current is high, which the sense amplifier interprets as a logical 1.
    - **OR Operation:** Any cell in the low-resistance state will generate a high output on the bitline, resulting in a logical OR operation.
  - **Parallelism:** ReRAM can perform these operations in parallel across the entire crossbar array, accelerating tasks that require significant data processing, such as neural network inference.
- Matrix-Vector Multiplication
  - **Crossbar Array Activation:** To perform matrix-vector multiplication, ReRAM arrays are configured such that each row in the matrix corresponds to a wordline, and each element of the vector is represented as an input voltage on the bitlines.
  - **Ohm's Law and Kirchhoff's Law:** By applying voltages to specific wordlines, the currents generated by each ReRAM cell follow Ohm's Law (V = IR). The combined current on a bitline (following Kirchhoff's Current Law) represents the sum of products for the elements in a row of the matrix and the vector.
  - **Analog Summation of Currents:** The sense amplifiers on each bitline detect and sum the analog currents, effectively performing the matrix-vector multiplication directly within the memory array.

- Analog Operations
  - **Analog Current Accumulation:** ReRAM cells' ability to store multiple resistance levels makes them ideal for analog computation. By encoding data as resistance values, the current output from each cell represents a weighted value.
  - **Multi-Level Cell (MLC) Support:** ReRAM can store multiple bits in a single cell, enabling finer granularity in resistance levels and allowing for analog summation of values.
  - **Tolerating Noise:** While analog operations in ReRAM can introduce noise, they're effective for applications that can tolerate approximation, such as machine learning, where exact values aren't always necessary. Multiple rows can be activated to sum analog values across the bitlines, leveraging ReRAM's resistance characteristics to process large-scale data.

D. Flash Memory-based IMC

Flash memory-based in-memory computing (IMC) uses the characteristics of NAND or NOR flash memory cells to enable computation directly in memory.

1) Flash Memory Architecture Basics: Flash memory is a type of non-volatile storage that retains data without power. Flash cells store data using floating-gate transistors, where charges trapped on a floating gate determine the cell's threshold voltage and represent binary data (0 or 1). In multi-level cells (MLCs), different voltage thresholds can store multiple bits per cell, which is useful for analog or approximate computing.

Flash memory is commonly arranged in:

- **NAND Flash Arrays:** Used for high-density data storage with slower read speeds but higher density.
- **NOR Flash Arrays:** Faster read speeds and is often used where random access is needed.

2) Functioning of Flash Memory-based IMC: In an IMC setup, the ReRAM architecture is adapted to perform computational operations directly in memory, leveraging its unique resistive characteristics for both storage and processing.

- **Modified Periphery Circuits**: In flash-based IMC, threshold detection allows flash cells to represent different logic states based on their threshold voltage, with sense amplifiers interpreting the results for both bitwise and analog operations. Charge-sharing techniques enable voltage-level interpretation across cells, supporting logic and summation tasks.
- **Row/Column Activation**: Flash memory supports parallel row activation, allowing multiple rows to contribute to computation simultaneously, which is ideal for matrix multiplication. Selective voltage application to wordlines and bitlines facilitates specific operations like multiplication and bitwise logic (AND, OR, XOR).
- **Data Encoding**: Flash memory stores binary data with high and low threshold voltages for precise bitwise operations. It also supports multi-level cell (MLC) encoding, where multiple threshold levels allow for approximate or analog computations, ideal for applications like machine learning where precise values are not always necessary.
- **Sense Amplifiers and Computation Techniques**: In analog computations like matrix-vector multiplication, sense amplifiers detect the combined threshold voltage from multiple activated rows, representing a sum of values. Voltage manipulation of wordlines enables bitwise operations (AND, OR, XOR), with sense amplifiers interpreting the results for tasks like cryptographic functions and machine learning inference.

3) Flash Memory for In-Memory Computing (IMC):

- Bitwise Operations (AND, OR, XOR)
  - **Voltage Thresholds for Logic Operations:** Specific threshold voltages represent binary states. Bitwise operations (AND, OR, XOR) are performed by applying specific voltages to wordlines and bitlines, leveraging the resistance states.
  - **Simultaneous Activation for Parallel Processing:** By activating multiple cells simultaneously, flash IMC can perform bitwise operations across rows and columns, useful in data processing tasks.
- Matrix-Vector Multiplication
  - **Data Mapping:** Matrix data is mapped to the flash array with each row representing elements of the matrix, and input vector values are applied as voltages.
  - **Crossbar Summation Using Sense Amplifiers:** When multiple rows are activated, the combined threshold voltages contribute to a current summation across the bitlines. The sense amplifiers interpret these currents, providing partial products necessary for matrix-vector multiplication.
- Analog Operations
  - **Analog Encoding in Multi-Level Cells:** Flash cells with multiple threshold voltages can store a range of values. This encoding allows approximate computations, like neural network inference, where exact values are not critical.
  - **Summation with Charge Sharing:** By activating multiple cells, flash IMC achieves analog summation where the cumulative charge or voltage levels on the bitlines represent a summed value across rows, useful in analog computations.

E. Hybrid IMC

Some IMC designs combine different memory technologies (e.g., SRAM and DRAM) to leverage the strengths of each type. These hybrid architectures aim to balance performance, scalability, and power efficiency, adapting to a wide range of computational requirements.

1) Hybrid Architecture Basics: Hybrid-based IMC combines SRAM, DRAM, and ReRAM, utilizing each memory type's unique strengths.

- **SRAM:** Allows rapid access and low power consumption, making it suitable for cache or buffer applications.
- **DRAM:** It's often used for intermediate storage and computation due to its balance of speed and density.
- **ReRAM:** It can retain data without power and supports both binary and multi-level storage, ideal for low-power and persistent storage applications.

2) Hybrid Architecture for In-Memory Computing (IMC):

- Bitwise Operations (AND, OR, XOR)
  - **Threshold Voltage for Logic Operations:**
    - **SRAM:** High-speed bitwise operations like AND, OR, and XOR through rapid row activation.
    - **DRAM:** Similar operations with slower response, used for larger datasets.
    - **ReRAM:** Adjustable resistance states enable approximate bitwise logic, advantageous for low-power and low-precision tasks.
  - **Parallel Processing Capability:**
    - SRAM excels in high-speed parallel processing.
    - DRAM and ReRAM process data with higher density, beneficial for tasks like database searching and encryption.
- Matrix-Vector Multiplication
  - **Data Mapping:** Matrix elements are mapped across different memory arrays:
    - SRAM processes elements requiring high-speed computation.
    - DRAM handles denser matrices where access time is less critical.
    - ReRAM stores multi-bit representations for approximations and larger datasets.
  - **Crossbar Summation Using Sense Amplifiers:**
    - SRAM enables fast summation with low latency.
    - DRAM supports more extensive data operations with moderate speed.
    - ReRAM leverages multi-level cells, offering analog summation suitable for approximate computations.
- Analog Operations
  - **Analog Summation and Charge Sharing:** DRAM and ReRAM both support analog summation techniques. By applying appropriate voltages, DRAM achieves temporary charge accumulation, while ReRAM's multi-level cells allow stable analog encoding.

VI. Simulators

We conducted a thorough investigation into existing open-source simulators for In-Memory Computing (IMC). The goal was to identify simulators that are widely recognized in academic and research communities and capable of effectively simulating IMC architectures. Our selection criteria included functionality, documentation quality, scalability, performance metrics supported, and ease of integration into our benchmarking environment.

1) PUMA
2) PIM-SIM
3) Cross-SIM
4) MN-SIM
5) Multi-SIM
6) CIM-SIM

VII. Study of Model Architectures of Simulators Selected

A. PUMA

The PUMA (Processing Using Memory Architecture) In-Memory Computing simulator is designed to simulate machine learning tasks, focusing on minimizing the data movement bottlenecks by integrating memory and processing units.

1. Memory-Processing Integration: PUMA uses SRAM cells for both memory storage and processing, eliminating the need for traditional data transfer between CPU and memory. This integration reduces latency and power consumption. Analog computation, such as matrix-vector multiplications, is performed within the memory cells, optimizing the execution of machine learning tasks, especially for multiply-accumulate (MAC) operations.

2. Processing Elements (PEs) and Local Computation: The SRAM cells are organized into arrays of Processing Elements (PEs), which can execute operations directly on the data stored in memory. These PEs are equipped with local accumulators and shift registers to handle intermediate results without needing to access external memory, enhancing the efficiency of neural network operations.

3. Analog-Digital Partitioning: PUMA incorporates Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs) to facilitate seamless interaction between analog and digital operations. Analog computation is used for efficient MAC operations, while digital processing handles precise control flow and computation. Multi-level cells (MLCs) store multiple bits of data per cell, supporting complex analog calculations.

4. Hierarchical Interconnects and Data Flow Optimization: The architecture features a hierarchical interconnect design, optimizing data flow between memory cells and processing units. Local and global buses manage high-throughput data communication efficiently, minimizing latency and congestion, which is essential for large-scale computations in machine learning.

5. In-Memory Acceleration for Neural Networks: PUMA accelerates neural network computations by performing MAC operations within the memory arrays, allowing efficient execution of convolutional and fully connected layers. Parallelism and pipelining techniques are used to handle multiple layers of neural networks simultaneously, improving computational throughput.

B. PIM-SIM

PIM (Processing-In-Memory) SIM is another In-Memory Computing (IMC) simulator, and its design focuses on reducing the bottleneck created by data movement between the processor and memory in traditional computing systems. Unlike traditional systems where the CPU and memory are separate, PIM SIM integrates processing elements directly within the memory to enable computation at the memory level. This leads to significant improvements in performance and energy efficiency for memory-bound tasks, including machine learning, data analytics, and scientific computations.

1. Memory-Processing Integration: PIM SIM integrates processing elements directly within memory chips, using PIM-enabled DRAM modules. This allows computations to be executed inside the memory, reducing the need for data transfer between the processor and memory. Each memory bank contains a simple processing unit (such as an ALU) that can directly perform arithmetic and logical operations on data stored within the memory, enhancing efficiency.

2. Processing Elements (PEs) and Local Computation: Each memory bank in PIM SIM houses processing elements (typically small ALUs) that can perform operations like addition and multiplication. This enables local data manipulation without the need for data fetching from external memory, improving the execution of memory-bound tasks such as matrix multiplications and vector additions.

3. Parallelism and Distributed Processing: PIM SIM supports parallel execution by distributing tasks across multiple memory banks, each with its own processing unit. This enables high data-level parallelism, allowing operations to be performed on different portions of a dataset simultaneously, which accelerates tasks like machine learning and data analytics.

4. Interconnects and Data Flow: The memory banks are connected to the processing elements via high-speed local interconnects, allowing fast data access. For tasks requiring data from multiple banks (e.g., matrix multiplication), cross-bank communication is facilitated by a high-bandwidth interconnect network, optimizing data flow.

5. Task Offloading and Specialization: PIM SIM excels at offloading repetitive and memory-intensive tasks, such as matrix operations and data filtering, directly to memory. This reduces latency and improves throughput, particularly for machine learning workloads like matrix-vector multiplication, which can be processed directly in memory.

C. Cross-SIM

Cross-Sim is a high-performance simulation tool designed for benchmarking and evaluating complex computing architectures, including In-Memory Computing (IMC) systems. While not inherently aimed at IMC, its flexible architecture allows for simulating IMC systems by incorporating custom memory units with processing elements.

1. Architecture Overview: Cross-Sim is a high-performance simulation tool designed for benchmarking and modeling complex systems. It supports system-level simulations with modular components for memory, processors, I/O, and network. These components interact within a flexible framework, enabling simulations of large-scale, heterogeneous systems, including In-Memory Computing (IMC).

2. Achieving IMC: Cross-Sim simulates IMC by integrating processing elements within memory units. This includes modeling processing-in-memory (PIM) systems, where computational tasks are executed inside memory rather than transferring data to external processing units. It also supports task-level parallelism and memory-parallelism, reducing data movement and latency.

3. Emerging IMC Technologies: Cross-Sim supports the simulation of advanced IMC technologies like 3D-stacked memory and near-memory processing. These architectures allow for reduced latency and increased bandwidth, essential for efficient IMC operations. Cross-Sim can model memory units with embedded processing capabilities, simulating how tasks are executed directly within the memory subsystem.

4. Energy Efficiency: Cross-Sim includes energy profiling tools to assess power consumption in IMC systems. By simulating different memory-processing integration schemes, it allows users to evaluate energy efficiency and identify configurations that minimize power usage. This capability is crucial for designing low-energy IMC architectures.

5. Fault Tolerance: Cross-Sim models error detection and correction mechanisms within memory units, such as Error-Correcting Codes (ECC). It can simulate failure scenarios like bit flips or memory cell failures, providing insights into the fault tolerance and reliability of IMC systems.

6. High-Performance IMC Simulation: Designed for large-scale simulations, Cross-Sim is ideal for high-performance IMC modeling. It supports distributed simulations and can scale to simulate complex, data-intensive IMC systems. This makes it well-suited for evaluating performance, memory bandwidth, and latency in IMC architectures.

D. MN-SIM

MNSIM (Memory Network Simulator) is a simulator used for modeling and simulating memory network systems, particularly those focused on memory-centric computing architectures such as In-Memory Computing (IMC).

1. Memory-Processing Integration: MNSIM integrates processing engines directly within memory units, enabling computations like matrix multiplications and search algorithms to be performed within memory itself. This reduces the data movement bottleneck seen in traditional computing systems, improving performance for memory-intensive tasks such as data analytics and machine learning.

2. Memory Network Architecture: MNSIM employs a distributed memory system, where data is organized across multiple memory banks. Each memory bank contains dedicated processing elements (PEs) capable of executing operations locally on the data. These memory banks are interconnected via a high-bandwidth network, allowing efficient cross-bank communication for complex operations that involve data from multiple banks.

3. Parallel Processing and Data Locality: MNSIM supports high parallelism by allowing multiple memory banks to operate on different data chunks concurrently. Each PE works on local data within its memory bank, reducing latency. Data locality is also optimized with caching mechanisms that ensure frequently accessed data remains close to the processing units, further minimizing access delays.

4. Local and Global Data Processing: Local memory operations, such as vector additions and matrix multiplications, are executed directly within each memory bank. For global operations requiring data from multiple banks, MNSIM uses its network to facilitate data exchange, minimizing the power and time costs typically associated with external data transfers.

5. Efficient Memory Access and Energy Efficiency: Since computations occur within memory, MNSIM eliminates the need for data transfers between processors and memory, significantly reducing latency. This architecture is energy-efficient, as it minimizes the power consumption typically involved in data movement across traditional systems.

6. Hardware-Software Co-Design: MNSIM allows for the simulation of both hardware (memory-centric units and processing elements) and software optimizations, such as task scheduling and parallelization strategies. This helps maximize the efficiency of in-memory processing by modeling how software interacts with the hardware.

E. Multi-SIM

Multisim is primarily an electronics simulation software that supports both analog and digital circuit design. While not inherently designed for In-Memory Computing (IMC), it can be adapted for simulating certain hardware interactions relevant to IMC principles.

1. Component-Based Simulation: Multisim utilizes a component-based simulation approach for both analog and digital systems. It includes a wide range of components like resistors, capacitors, logic gates, microcontrollers, and custom memory elements (e.g., RAM, ROM). Custom processing elements can be integrated within memory models to simulate Processing-in-Memory (PIM) architectures.

2. Memory-Processing Integration: While Multisim doesn't directly simulate in-memory processing, users can design custom memory units with embedded processing elements. This allows simulations where simple arithmetic operations (e.g., adders, multipliers) occur within the memory block itself, reducing the need for external processors. This setup can be used to model PIM or PUMA systems.

3. Parallelism and Low-Latency Access: Multisim supports parallel circuit simulations, which could represent parallel memory accesses in IMC systems. Memory banks with integrated processing units can perform simultaneous operations, mimicking the low-latency, data-locality principles in IMC.

4. Energy Efficiency: Multisim offers energy consumption modeling for circuits. By simulating power draw and current usage within memory-processing integrated systems, it allows researchers to estimate the energy savings of in-memory computations.

5. Fault Tolerance: Error-checking components (e.g., voltage fluctuations, component failure) can be simulated within Multisim, though it doesn't directly model advanced error correction techniques like ECC. However, these basic checks can be extended to simulate reliability in hybrid memory-processing systems.

F. CIM-SIM

CIM-Sim (Compute-In-Memory Simulator) is a specialized simulation tool designed to model In-Memory Computing (IMC) systems, where computation is performed directly within memory units. By integrating processing capabilities into memory, CIM-Sim enables researchers to simulate and evaluate the performance of CIM architectures, which reduce latency and improve energy efficiency.

1. Architecture Overview: Cross-Sim is a high-performance simulation tool designed for benchmarking and modeling complex systems. It supports system-level simulations with modular components for memory, processors, I/O, and network. These components interact within a flexible framework, enabling simulations of large-scale, heterogeneous systems, including In-Memory Computing (IMC).

2. Achieving IMC: Cross-Sim simulates IMC by integrating processing elements within memory units. This includes modeling processing-in-memory (PIM) systems, where computational tasks are executed inside memory rather than transferring data to external processing units. It also supports task-level parallelism and memory-parallelism, reducing data movement and latency.

3. Emerging IMC Technologies: Cross-Sim supports the simulation of advanced IMC technologies like 3D-stacked memory and near-memory processing. These architectures allow for reduced latency and increased bandwidth, essential for efficient IMC operations. Cross-Sim can model memory units with embedded processing capabilities, simulating how tasks are executed directly within the memory subsystem.

4. Energy Efficiency: Cross-Sim includes energy profiling tools to assess power consumption in IMC systems. By simulating different memory-processing integration schemes, it allows users to evaluate energy efficiency and identify configurations that minimize power usage. This capability is crucial for designing low-energy IMC architectures.

5. Fault Tolerance: Cross-Sim models error detection and correction mechanisms within memory units, such as Error-Correcting Codes (ECC). It can simulate failure scenarios like bit flips or memory cell failures, providing insights into the fault tolerance and reliability of IMC systems.

6. High-Performance IMC Simulation: Designed for large-scale simulations, Cross-Sim is ideal for high-performance IMC modeling. It supports distributed simulations and can scale to simulate complex, data-intensive IMC systems. This makes it well-suited for evaluating performance, memory bandwidth, and latency in IMC architectures.

VIII. Benchmarking Programs

To ensure comprehensive and fair evaluation, we employ custom programs designed specifically for this project. These programs reflect both general-purpose and IMC-specific workloads, providing a detailed analysis of each simulator's strengths and limitations.

1) **Matrix Multiplication (MatMul)** - A fundamental operation in many algorithms, including deep learning, signal processing, and scientific computing. It's widely used to test the compute-memory balance in IMC systems.
2) **Neural Network Inference** - Run inference on simple neural networks (such as Fully Connected Networks) to test how well the IMC system handles memory-intensive operations and matrix-vector multiplications.
3) **Convolutional Neural Networks (CNNs)** - Particularly layers like 2D convolutions, max-pooling, and ReLU operations. This can be adapted from models like AlexNet or ResNet to test how memory bandwidth and access latency affect performance.
4) **Long Short-Term Memory (LSTM) Networks** - Used for sequence prediction tasks, LSTMs have heavy memory access patterns due to recurrent connections and are ideal for testing sequential memory access in IMC architectures.
5) **Sparse Matrix-Vector Multiplication (SpMV)** - Frequently used in scientific computing and machine learning. Sparse matrices stress memory access patterns, benchmarking how IMC handles memory-dominant operations.
6) **DNN Training (Backpropagation)** - Perform backpropagation training on a small neural network to assess how well IMC systems handle the combination of memory-intensive weight updates and compute-heavy gradient calculations.

These programs provide a good mix of memory and compute-intensive tasks, making them suitable for benchmarking on IMC simulators.

IX. Performance Metrics

The performance metrics were selected based on their relevance to In-Memory Computing (IMC) and the specific workloads being tested. Since IMC systems prioritize reduced data movement and energy-efficient computation, we focused on metrics that highlight these features while also considering standard performance measurements such as execution time and throughput.

1) **Execution Time** - The total time taken by the simulator to complete a given benchmark, from initialization to output.
2) **Memory Usage** - The amount of memory consumed by the simulator during the execution of a benchmark.
3) **Latency** - The delay between the initiation of a task (input) and the completion of that task (output).
4) **Throughput** - The amount of data processed per unit of time, typically measured in operations per second.
5) **Energy Efficiency** - Measured in joules per operation or task, indicating energy consumption for completing a task.
6) **Scalability** - The ability of the simulator to maintain performance as the size of the dataset or the number of parallel operations increases.

X. Benchmarking Process

During the benchmarking process, we encountered significant compatibility and setup issues, limiting our ability to fully execute all planned experiments. Specifically, we successfully ran only three models, while three others failed due to compatibility and setup challenges, as outlined below:

- **Successful Runs:**
  - For MNSIM, we were able to run AlexNet, LeNet, and ResNet18 models.
  - For PUMA, only the LSTM and MLP programs executed successfully due to compiler dependencies specific to these models.
  - For PIMSIM, ResNet18 was the only model that ran successfully; other models failed during the ONNX-to-simulator format conversion stage.
- **Unsuccessful Runs:**
  - Compiler issues, dependency conflicts, and simulator-specific requirements prevented the execution of other programs across the selected simulators.
  - Some simulators required outdated tools or configurations incompatible with modern systems, as highlighted in the challenges section.

These issues emphasize the lack of standardization in benchmarking tools and simulators, which hinders comprehensive evaluation. The programs that successfully ran represent the only subset we could reliably benchmark under the given constraints.

XI. Result, Analysis and Discussion

Latency (ms)

Framework	Model	Latency (ms)
MNSIM	AlexNet	0.8248
	LeNet	0.35549
	ResNet18	3.087431
	vgg8	5.330703
	vgg16	3.593919
PUMA	LSTM	NA
	MLP	NA
PIMSIM	ResNet18	25

Power (W)

Framework	Model	Power (W)
MNSIM (Hardware Power)	AlexNet	108.1671
	LeNet	3.2541
	ResNet18	498.5991
	vgg8	338.6245
	vgg16	534.9752
PUMA (unitless)	LSTM	42637.4632
	MLP	4543.8574
PIMSIM (Chip power)	ResNet18	0.0659 (mW)

Area (μm^2)

Framework	Model	Area (μm^2)
MNSIM (Hardware Area)	AlexNet	437472193.166
	LeNet	57239352.377
	ResNet18	2015642908.697
	vgg8	1316505104.666
	vgg16	2101501937.263
PUMA (Node Area unitless)	LSTM	28145.878
	MLP	28145.878
PIMSIM	ResNet18	NA

Energy (nJ or J)

Framework	Model	Energy
MNSIM (Hardware Energy)	AlexNet	313829.1099 nJ
	LeNet	7906.1866 nJ
	ResNet18	935655.5399 nJ
	vgg8	3827427.1532 nJ
	vgg16	1923761.6899 nJ
PUMA (Node Energy)	LSTM	0.0041 J
	MLP	0.000124 J
PIMSIM	ResNet18	1647884735 pJ/it

Simulation Time (s)

Framework	Model	Simulation Time (s)
MNSIM	AlexNet	0.3910
	LeNet	0.1142
	ResNet18	1.5233
	vgg8	0.9770
	vgg16	1.6312
PUMA (unitless)	LSTM	9.5583e-05
	MLP	2.7274e-05
PIMSIM	ResNet18	0.1

Analysis

While benchmarking provided insights into the performance of individual simulators, we were unable to perform a comprehensive comparison across all simulators due to the lack of standardized programs that could run on each. Compatibility issues and varying dependencies restricted our ability to benchmark all simulators using the same set of tasks, as detailed in the challenges section. This limitation highlights the urgent need for standardized tools and frameworks in the IMC research domain to enable meaningful comparisons.

Architectural Comparison

The architectural differences between the six simulators are summarized below, showcasing their unique design philosophies and target applications.

TABLE I
Architectural Comparison of IMC Simulators

Simulator	Memory Type	Key Features	Applications
PUMA	SRAM	Analog-digital integration, MLP	Neural networks
PIMSIM	DRAM	Processing-in-memory, ONNX support	Data analytics, AI/ML
MNSIM	SRAM/Hybrid	Bitwise ops, matrix multiplication	General-purpose workloads
CIM-SIM	Flash/ReRAM	Non-volatile memory, analog ops	Low-power AI/ML
Multi-SIM	SRAM	Customizable processing elements	Analog/digital circuit design
Cross-SIM	Hybrid	Modular components, scalability	HPC, heterogeneous systems

This comparison provides a high-level overview of the architectures, emphasizing their distinct capabilities and the specific niches they cater to. Despite their strengths, interoperability and consistent benchmarking remain critical challenges for IMC simulators.

XII. Challenges Faced

1) For PUMA, which is Python 2.7-dependent, a virtual environment was used to overcome compatibility issues, allowing the execution of MLP and LSTM models. However, the compiler required specific C++ and configuration files only available for these two models, which limited usability.
2) In PIMSIM, a compiler was available to convert models from ONNX format to the desired format, and ResNet18 conversion was successfully completed. However, no output was generated when compiling other models, presenting unresolved issues regarding compatibility or output handling.
3) Selecting open-source simulators was challenging due to limited options meeting desired functionality and support criteria. The availability of well-documented and robust simulators posed a significant barrier to the benchmarking process.
4) Simulators that were successfully executed supported only a subset of programs and lacked consistency. To ensure fair and reliable benchmarking, it was essential

to run the same program set across all simulators, which was not feasible with the available tools.

XIII. Open Issues

1) CrossSim provides a library-like interface but does not model analog accelerator attributes such as energy, area, or speed. It also requires CUDA for operation, making it incompatible with AMD systems.
2) CIMulator lacks proper documentation, making implementation and functionality difficult to assess.
3) MultiSim does not have a repository, leaving a gap in accessibility and evaluation.
4) Due to the inability to run several simulators and the lack of consistency across those that were executed, a comprehensive benchmarking comparison could not be performed.
5) Standardization of benchmarking tools and datasets across simulators remains an unresolved issue, limiting the scope for comparative analysis and evaluation of these tools.

XIV. Summary and Takeaway

In this project, we conducted an architectural study and benchmarking of various open-source In-Memory Computing (IMC) simulators. Through detailed analysis, we highlighted the strengths and limitations of each tool in terms of usability, compatibility, and performance.

The key takeaways from our study include:

- Open-source IMC simulators play a crucial role in exploring innovative computing architectures but are often limited by inadequate documentation, restricted compatibility, and inconsistent support for benchmarking.
- Effective benchmarking requires simulators to run a common set of programs under standardized conditions, which is a significant limitation in the current ecosystem.
- Future work should focus on improving simulator documentation, increasing cross-platform compatibility, and standardizing benchmarking protocols to foster more consistent and meaningful evaluations.

This study serves as a guide for researchers and engineers in selecting appropriate IMC simulators while emphasizing the need for further advancements in this domain.

XV. Author Contribution

- **Vraj Shah (22110292):** Conducted in-depth research on In-Memory Computing (IMC) principles and architectures. Defined key benchmarking programs and performance metrics.
- **Shubham Agrawal (22110249):** Compiled and structured the report. Contributed to the comparative analysis of simulators and wrote sections on open issues, challenges faced, and summary.
- **Dewansh Singh Chandel (22110072):** Performed benchmarking experiments across selected simulators and analyzed results. Provided insights into simulator usability and limitations.
- **Sawale Sumeet Shivaji (22110234):** Focused on simulator selection and configuration. Ran experiments on PUMA and PIMSIM and documented challenges related to their execution.

References

[1] M. E. Fouda, T. Tuma, and M. Le Gallo, "An Overview of Computation-in-Memory (CIM) Architectures," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 10, pp. 3271–3284, Oct. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9151594

[2] A. A. Khan, Z. Liu, N. Talati, and S. Aga, "The Landscape of Compute-near-memory and Compute-in-memory: A Research and Commercial Overview," *ACM Computing Surveys*, vol. 54, no. 2, Article 22, May 2022. [Online]. Available: https://doi.org/10.1145/3424624

[3] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "In-Memory Computing: Advances and Prospects," *Nature Nanotechnology*, vol. 15, no. 7, pp. 529–544, Jul. 2020. [Online]. Available: https://doi.org/10.1038/s41565-020-0677-6

[4] B. Li, E. Ipek, C. H. Kim, and J. Kim, "An Overview of In-memory Processing with Emerging Non-volatile Memory for Data-intensive Applications," *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1209–1232, Aug. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9131347

[5] J. Park, S. Kim, and H. Kim, "In-Memory Associative Processing: Review and Outlook," *IEEE Micro*, vol. 39, no. 3, pp. 14–23, May 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8714756

[6] A. Ankit, I. El Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-M. Hwu, J. P. Strachan, K. Roy, and D. Milojicic, "PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference," *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019. [Online]. Available: https://arxiv.org/abs/1901.10351

[7] A. Ankit, P. Silveira, and G. Aguiar, "PUMA-Simulator: A Detailed Simulation Model of a Dataflow Architecture Built with NVM," 2019. [Online]. Available: https://github.com/Aayush-Ankit/puma-simulator

[8] S. Xu, X. Chen, Y. Wang, Y. Han, X. Qian, and X. Li, "PIMSim: A Flexible and Detailed Processing-in-Memory Simulator," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 29–32, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8567968

[9] S. Xu, X. Chen, Y. Wang, Y. Han, X. Qian, and X. Li, "PIMSim: A Flexible and Detailed Processing-in-Memory Simulator," 2019. [Online]. Available: https://github.com/vineodd/PIMSim

[10] Y. Chen, T. Tang, S. Liu, and Y. Wang, "MNSIM 2.0: A Behavior-Level Modeling Tool for Processing-In-Memory Architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 1, pp. 1–14, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10058114

[11] X. Wang, X. Sun, Y. Han, and X. Chen, "PIMSIM-NN: An ISA-based Simulation Framework for Processing-in-Memory Accelerators," 2024. [Online]. Available: https://arxiv.org/abs/2402.18089

[12] J. Chen, J. Gómez-Luna, I. El Hajj, Y. Guo, and O. Mutlu, "SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory," 2023. [Online]. Available: https://arxiv.org/abs/2310.01893

[13] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016. [Online]. Available: https://github.com/CMU-SAFARI/ramulator-pim