electronics

MDPI

Article

PIMCoSim: Hardware/Software Co-Simulator for Exploring Processing-in-Memory Architectures

Jinyoung Shin , Seongmo An , Sangho Lee and Seung Eun Lee *

Department of Electronic Engineering, Seoul National University of Science and Technology, Seoul 01811, Republic of Korea; shinjinyoung@seoultech.ac.kr (J.S.); ahnseongmo@seoultech.ac.kr (S.A.); leesangho@seoultech.ac.kr (S.L.)

* Correspondence: seung.lee@seoultech.ac.kr; Tel.: +82-2-970-9021

Abstract: As the scope of artificial intelligence (AI) expands and the structure becomes more complex, the amount of data for inference and training has increased. In traditional computer architectures, the memory bandwidth limitations have intensified bottlenecks in AI systems, and processing-in-memory (PIM) architectures have been proposed to overcome this issue. PIM is an architecture that performs computations within memory, thereby reducing data movement between the CPU and memory. However, since PIM is difficult to optimize as a general-purpose architecture, it is essential to adopt an architecture suitable for the target application. While various simulators and emulators have been introduced for the design space exploration (DSE) of different PIM architectures, simulators are limited in debugging hardware operations, and emulators face challenges in flexibly modifying the system configuration, as emulators implement the entire architecture in hardware. Therefore, this paper introduces PIMCoSim, a comprehensive hardware–software co-simulator for the DSE of DRAM-PIM systems. This co-simulator partially emulates simplified hardware-implemented processing elements (PEs) and integrates software models for memory operations, facilitating the DSE of PIM systems. To validate PIMCoSim, we analyzed results for different computational workloads by varying PIM structures and operational policies, demonstrating the efficiency of DRAM-PIM systems. The co-simulation approach in PIMCoSim aims to contribute to analyzing DRAM-PIM configurations and adopting optimized structures.

Keywords: artificial intelligence (AI); co-simulator; dynamic random-access memory (DRAM); processing in memory (PIM)

check for updates

Citation: Shin, J.; An, S.; Lee, S.; Lee, S.E. PIMCoSim: Hardware/Software Co-Simulator for Exploring Processing-in-Memory Architectures. *Electronics* **2024**, *13*, 4795. https://doi.org/10.3390/electronics13234795

Academic Editor: David Defour

Received: 12 November 2024
Revised: 2 December 2024
Accepted: 2 December 2024
Published: 5 December 2024

1. Introduction

As AI is applied to various applications, significant research is being conducted on the architecture of AI [1–3]. With the increasing complexity of neural network architectures, implementing AI requires managing an enormous number of parameters, which leads to increased data movement between memory and processors and results in repetitive tasks. The growth in maximum floating-point operations per second of hardware has exceeded the growth in DRAM bandwidth, widening the gap between the two components. Consequently, traditional computer architectures, composed of CPUs and memory, struggle to meet the rising demands for data processing [4]. This challenge causes system performance degradation and bottlenecks, eventually leading to what is known as the memory wall.

To address these challenges, research has been conducted to develop novel computer architectures. In the von Neumann architecture, where instructions and data are stored in memory without physical separation, memory bandwidth causes bottlenecks. In response, the Harvard architecture was introduced, with physical separation of instruction and data spaces to enhance processing speed more efficiently.

A large body of research has focused on improving throughput by integrating multiple accelerators, graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs) into computer architectures, with particular attention given to parallel computing [5,6]. The GPUs play critical roles in high-performance computing tasks such as graphics processing, computer vision, and AI [7]. On the other hand, FPGAs and ASICs operate as accelerators by executing computations through optimized circuit architectures designed for specific applications, offering greater power efficiency compared to GPUs [8].

Research has also focused on addressing the memory wall caused by increased data movement by modifying memory architecture. Research study [9] utilizes the internal bus of dynamic random-access memory (DRAM) to transfer data from the source row to the target row in a pipelined manner, enabling the efficient usage of large volumes of data, specialized instructions are defined to manage large data operations, and small logic additions are incorporated to facilitate efficient execution of these tasks. Another promising approach is resistive random-access memory (ReRAM), which leverages resistance state changes due to external stress to store data. ReRAM offers high read and write speeds, a simple structure, and high integration density. These features make ReRAM an attractive option for high-performance computing applications, such as machine learning. The unique features of ReRAM have sparked growing interest in its potential to meet the demands of data-intensive applications [10].

Data movement bottlenecks arise when all computations are handled exclusively by the arithmetic and logical units (ALUs) of a CPU. In response, PIM architectures have been introduced to perform computations closer to or within memory, reducing data transfers between the CPU and memory [11]. However, as PIM cannot deliver optimized performance with a single architecture across all applications, it is crucial to adopt a structure suited to specific applications. To enhance the performance of PIM architectures, various research efforts are being conducted, exploring options such as compiler optimizations, instruction set definitions, the number and functionality of PEs, cache architectures, data path structures, and the types of memory [12,13].

As research on PIM architectures progresses, the importance of simulation tools for effectively analyzing and improving various designs has become increasingly significant. Simulation tools play a critical role in establishing environments to evaluate and improve system performance by exploring and analyzing not only target systems but also PIM architectures alongside memory simulations [14–16]. Accordingly, simulators and emulators for the DSE of PIM architectures have been introduced. Software simulators model various components of PIM, including PIM kernels, memory timing, power consumption, and memory controller policies, allowing for the exploration of diverse configurations. Moreover, these tools offer the ability to modify various system configurations and enable the flexible simulation of the entire system by integrating PIM simulators with those of conventional subsystems.

However, software simulators face performance limitations when simulating complex systems, particularly for computation-intensive, heterogeneous computing architectures, and present challenges in accurately modeling system architecture [17]. To overcome these limitations, researchers have focused on implementing structurally accurate target models on FPGAs for emulation, aiming to achieve both speed improvements and reliable results. The emulator enhanced the simulation speed and provided a comprehensive analysis of the entire system, including the PIM architecture and cores. Furthermore, the enhanced speed enables researchers to perform DSE on PIM architectures with increasingly complex workloads.

Previous simulators, which run on CPU platforms, are not well suited for debugging the structural characteristics of hardware [18]. In contrast, emulators implement the entire PIM architecture in hardware, making it less flexible than simulators in terms of system configuration adjustments. Moreover, since emulators are designed for specific boards, implementing complex systems may present challenges for practical application. To address these issues, the need for co-simulation has emerged [19,20]. This approach combines software simulators and hardware emulators, enabling the verification of hardware's structural characteristics while also allowing for flexible simulation across various systems [21].

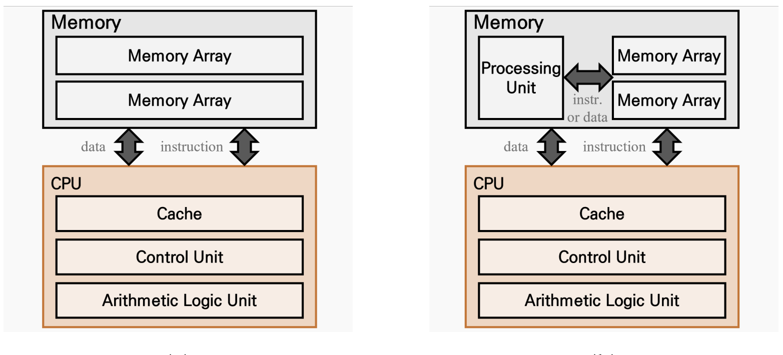
Compared to traditional simulators, where PIM's memory and processors are well detailed, allowing for complex simulations, there is a lack of precision in verifying the functional accuracy of PEs. PEs, which have a significant impact on the computational performance and throughput of PIM, must be analyzed through DSE to evaluate their performance and functionality, and to design structures suitable for specific systems and workloads. Therefore, we emphasize the need for a co-simulator capable of performing DSE on both PIM architectures and the hardware characteristics of PEs in a flexible environment.

In this paper, we propose PIMCoSim: an HW/SW co-simulator for exploring PIM architectures. PIMCoSim consists of a software simulator and hardware-implemented PEs of PIM, allowing researchers to implement DRAM-based PIM architectures in the simulator while simultaneously evaluating the hardware characteristics. The main contributions are as follows:

- The software simulator enables debugging of system operations by applying the modified PE operations and memory access policies to actual hardware implementations.
- Instead of emulating the entire system, PIMCoSim partially emulates simplified processing units, allowing for the examination of hardware characteristics and performance evaluation in a flexible environment.
- By utilizing the provided basic PIM-specific instruction set, applications are implemented in the software simulation, enabling the exploration of PIM designs suited to target workloads and instruction sets.

2. Background and Related Works

Modern data-intensive applications require higher frequencies and faster access than traditional memory systems. However, traditional memory systems are constrained by narrow bus widths, which limit performance, and frequent access results in inefficient energy consumption. As shown in Figure 1a, systems based on the von Neumann architecture experience bottlenecks due to bandwidth differences, limiting overall system performance. To address this issue, the concept of PIM, which combines RAM with computational elements, was introduced, followed by subsequent research utilizing DRAM [22].



(**a**) (**b**)

Figure 1. Comparison of traditional memory architecture (**a**) and PIM architecture (**b**).

The PIM architecture, as depicted in Figure 1b, performs computations directly in the memory, reducing not only the computational load on the CPU but also the data movement between the CPU and memory, making it suitable for alleviating bottlenecks. Since the computational units access the memory directly, PIM benefits from higher internal bandwidth. Additionally, PIM offers more efficient data processing by distributing the computational load between the memory and the CPU. PIM architectures with such advantages are classified based on the type and location of the computational units.

2.1. Data Array

In data arrays, computations are performed at the cell level of the memory, utilizing the analog properties of memory cells to support computational functions. In particular, extensive research has been conducted using resistive ReRAM.

ReRAM is a memory technology that stores data based on resistance states and demonstrates excellent performance in matrix–vector multiplication (MVM) operations. In the cell array of ReRAM implemented with a crossbar architecture, an input vector is applied as voltage across the word lines, generating currents proportional to the input voltage based on the state of each cell. According to Kirchhoff's law, the currents flowing through the bit lines are summed to perform MVM operations [23,24].

2.2. Subarray

In subarrays, computations are performed by connecting circuits that execute specific functions to the row buffer or sense amplifier of the memory, enabling computational capabilities. This approach leverages existing memory structures to process data directly within the memory, thereby reducing data movement and latency while enhancing overall system efficiency.

As shown in Figure 2d, DRAM is composed of cells made up of transistors and capacitors, where data are stored based on the charge state of each cell's capacitor. The DRAM subarray consists of components such as sense amplifiers and row buffers. The sense amplifier is positioned along each bitline of the subarray and amplifies the charge from the memory cells, ensuring stable data transfer. The row buffer temporarily stores the data amplified by the sense amplifier, improving data access speed when accessing consecutive rows. Research study [25] modifies the DRAM subarray to efficiently perform bitwise AND operations required for multiplication. By allocating a compute row to copy operands, activating the bitline precharge and the wordline for bitwise AND operations, the result is amplified through the sense amplifier connected to the bitline. Additionally, by integrating an adder tree into the row buffer, multiply–accumulate (MAC) operations are performed within the DRAM, reducing data movement overhead and enabling efficient data processing through parallel operations.

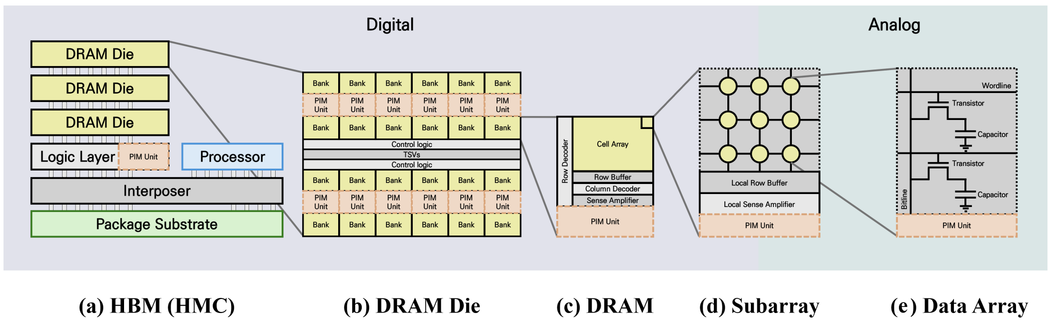


Figure 2. Overview of DRAM-PIM architecture.

2.3. Near Bank

Near memory banks, it is possible to integrate more advanced computing units compared to subarrays or data arrays, enabling various and complex operations and reducing the need to transfer data to the CPU. Research has been particularly active in memory systems based on DRAM where the near-bank processing approach enhances the computational efficiency of computer systems.

As depicted in Figure 2b, the DRAM die consists of multiple bank groups, each of which is further composed of individual banks. Adding computing units near a bank or bank group reduces memory traffic and enhances the utilization of internal memory bandwidth. Newton [26] places MAC units within DRAM banks to meet constraints on overall core area and power overhead. This enables PIM control through the DRAM interface and adds specific instructions for optimization. The placement of computation units within memory enables efficient parallel operations, achieving significant bandwidth improvements.

Additionally, by utilizing the stacking of DRAM, as presented in Figure 2a, high-bandwidth memory (HBM) was employed to achieve further bandwidth improvements. Research study [27] introduced an HBM-based PIM architecture that integrates single-instruction multiple data floating-point units (FPUs) near the bank. This architecture supports parallel processing at the bank level, categorized into single bank, all bank, and all bank PIM modes, and optimizes PIM operations by incorporating instruction register files, general register files, and scalar register files. By adding computation units suitable for machine learning tasks within DRAM banks, this approach saves energy and maximizes application performance.

2.4. PE Type

The implementation of PEs plays a critical role in PIM architectures, as throughput, area, and power characteristics vary depending on the design of the PE. Optimizing and allocating PEs to match the application and memory architecture enables a well-balanced PIM architecture. Research study [28] utilized a hybrid memory cube (HMC), where logic layers and multiple DRAM layers are vertically stacked through a hybrid memory cube (TSV), as shown in Figure 2a, to enhance execution efficiency in graph processing. Unlike HBM, HMC places specialized PEs for target operations in the logic layer, accelerating graph computations to improve speed and reduce energy consumption. Another approach is to use general-purpose processor cores instead of PEs specialized for specific operations. Research study [29] proposed memory-centric computing by dynamically scheduling tasks and data within the memory hierarchy to minimize data movement. By allocating data to memory and performing computations within the memory hierarchy, issues related to data movement, synchronization, and cache coherence are minimized. Implementing PEs on FPGAs with reconfigurable logic allows for flexible adjustment of computation units. Resarch study [30] targets convolutional neural networks (CNNs) and consists of PEs connected by a statically reconfigurable routing fabric, along with configurable logic blocks. Each block type is optimized based on specific applications, offering a more flexible system than PEs implemented as ASICs and enhancing computational performance.

2.5. Prior Related Works

As research on PIM architectures has advanced, the importance of simulators and emulators for modeling and analyzing architectural complexity and parallelism has grown, leading to the introduction of various designs. Table 1 summarizes simulators and emulators designed for the DSE of PIM architectures. Each tool is analyzed based on the system modeling level, distinctive featurs, and the simulation methods.

2.5.1. Simulator

PIMSim [31] integrates DRAMSim2 [32], HMCSim [33], and NVMain [34] simulators to support hybrid memory simulation, offering performance simulations with a tradeoff between speed and accuracy across three different modes. Additionally, gem5 [35] simulates the processor of the entire system, mimicking the core of the target PIM logic to support full-system-level research. The frontend of PIMSim recognizes and distributes PIM instructions, allowing researchers to define PIM kernels to minimize application constraints. Furthermore, cache hit rates and data locality are tracked, enabling adjustments to application partitioning strategies.

Sim^{2}PIM [36] provides flexible PIM simulation with high accuracy through compile-time instrumentation. By leveraging gem5, native code from various hosts, such as Intel, AMD, and ARM, suitable environments for PIM applications, is executed on the CPU, with support for hardware performance counters to provide reliable host-side information. These features allow for simulating only the PIM area, enhancing the simulation speed.

MultiPIM [37] anticipates that PIM will be employed in systems composed of multiple memory stacks rather than a single memory stack, meeting demands for higher memory capacity and bandwidth. Additionally, to reflect the architectural details of a real PIM system in the simulator, MultiPIM simulates specifics such as PIM core consistency, virtual memory, and latency between stacks. The simulator is divided into a frontend, which handles instruction simulation, and a backend, which handles memory access simulation, providing accurate memory request simulation.

CLAPPS [38] introduces a PIM simulator based on the HMC architecture. Employing a cycle-accurate approach, the simulator accurately models the memory and computational units within the HMC, enabling experimentation with various architectures. Supporting read–modify–write operations and parallel processing based on technical specifications of the HMC, the simulator is optimized for high-performance computing and big data applications. The simulator consists of independent modules, including transceivers, vaults, and a PIM interface, which support parallel simulation.

Table 1. Analysis of related works.

Source	System Modeling	Features	Simulation Method
PIMSim [31]	Full-system level, CPU, DRAM, HMC, NVM, PIM logic	Offers integration with memory-specific simulators, focusing on accurate memory system modeling at the DRAM level.	Cycle-accurate simulation into memory access patterns and PIM-specific instruction simulations.
Sim^{2}PIM [36]	Full-system level, CPU, memory	Emphasizes modularity and fast simulations by adopting configurable memory components to support agnostic simulations, focusing less on memory hierarchy specifics.	Fast, parallel simulation method to prioritize speed and modularity with host-side hardware performance counters for capturing real system metrics.
MultiPIM [37]	Full-system level, CPU, PIM logic, multi-stack DRAM	Targets multi-stack PIM systems, supporting complex multi-memory stack architectures and memory network timing through custom configurations.	Cycle-accurate simulation with instruction level and memory level for modeling timing in multi-stack-memory networks.
CLAPPS [38]	Full-system level, CPU, HMC, PIM logic	Focuses on HMC architecture with DRAM and 3D-stacked memory modeling, supporting detailed timing for various HMC components.	Cycle-accurate approach, implemented in SystemC to simulate the HMC architecture, aligning closely with the RTL design specifications of the HMC.
MEG [39]	Full-system level, CPU, HMC, nonvolatile memory, accelerator	A modular design with a bootable Linux image enables full-system software and hardware co-optimization.	Cycle-accurate RISC-V-based emulation platform with FPGA and HBM support, offering flexible performance monitoring.
PRIMO [18]	Memory-system level, DRAM, HBM, PIM logic	Full-stack PIM-DRAM framework with a specialized ISA and software stack for efficient memory management and cross-layer optimization, tailored for AI workloads.	Cycle-accurate, FPGA-based emulation approach that provides significant speed improvements over traditional CPU-based simulation, supporting detailed memory access trace collection and high-speed, real-time emulation.
PiMulator [40]	Memory-system level, DRAM, HBM, PIM logic	Integrates with the LiteX framework for flexible, full-system support on FPGAs, facilitating DSE, performance monitoring, and real-time emulation speed gains.	FPGA-based emulation to achieve cycle-accurate memory operations, maintaining emulation state and memory consistency across operation.

2.5.2. Emulator

MEG [39] is a cycle-exact system emulator developed utilizing HBM and an FPGA-based RISC-V processor. MEG collects full-system traces, provides hardware counter functionality for detailed system analysis, and emulates memory system behavior. MEG is well suited for verifying software flow and enables hardware–software co-optimization with Linux support.

PRIMO [18] is an emulator designed to overcome the execution time limitations of CPU-based PIM simulators and to emulate the operations of target DRAM-PIM architectures, allowing real-time workload analysis. Sixteen bank engines handle parallel operations to emulate DRAM-PIM on an FPGA, utilizing intra- and inter-bank memory timing controllers to provide accurate memory access timing information. The PIM compiler optimizes parallel processing by mapping scalar–vector and vector–matrix operations to the PIM domain. PRIMO provides memory access patterns and cycle information for target DRAM-PIM architectures, achieving speeds up to 6093 times faster than CPU-based simulators.

PiMulator [40] is an emulator that provides a soft-memory and soft-PIM infrastructure by implementing configurable memory and PIM models on FPGAs. PiMulator is implemented on an FPGA with a hierarchical architecture of chips, bank groups, banks, and subarrays to emulate various PIM architectures. Each level in the hierarchy features adjustable PIM logic, allowing for flexible configuration. PiMulator integrates with the LiteX framework, allowing for easy construction of PIM systems with RISC-V and various IPs. Timing is calculated based on an finite state machine (FSM) model of bank states, while a data synchronization engine synchronizes bank module data, efficiently utilizing the total memory capacity. PiMulator achieves an average speed that is 28 times faster than simulations using CPU-based gem5 and DRAMsim3 [41].

3. PIMCoSim

3.1. Overall Architecture

Figure 3 illustrates the target architecture of the PIM that the proposed PIMCoSim, the HW/SW co-simulator, aims to implement. Multiple pairs of static random-access memory (SRAM) caches and PEs are connected to the DRAM through the PE data path and PE controller.

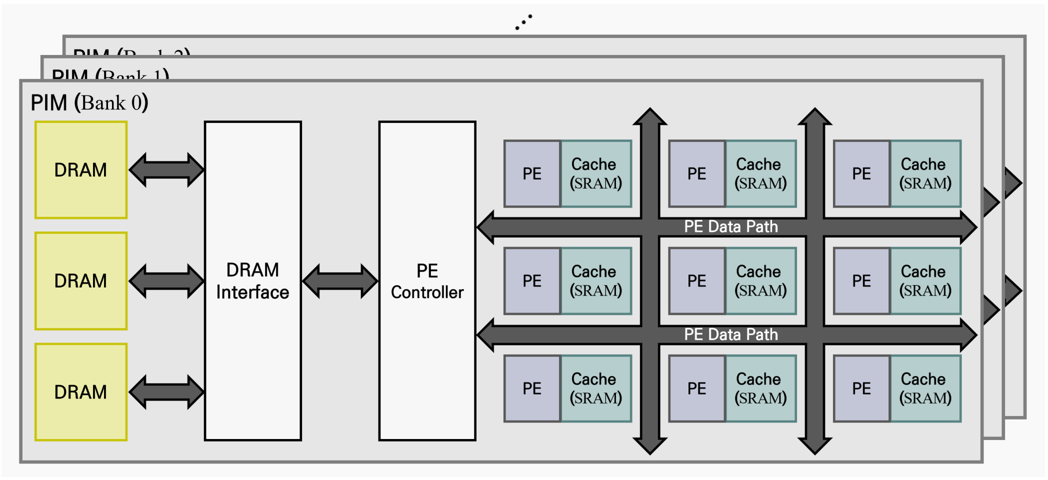


Figure 3. Targeted PIM architecture.

The DRAM and PE controller are simulated in software, while the actual hardware-implemented PEs are connected via serial communication. In the PIMCoSim architecture, unlike the targeted PIM architecture, multiple PEs are consolidated into a single PE for area efficiency. This single PE performs repeated operations as needed to achieve the same operational results as multiple PEs operating simultaneously. For the SRAM cache connected to the PE, a single SRAM's address space is divided into multiple spaces, making it as if each PE is connected to its own SRAM. The proposed PIMCoSim supports a maximum of nine PEs; thus 4 bits of the SRAM address are designated as the PE number to divide the address space.

3.2. Framework

Figure 4 illustrates the flow of hardware and software co-simulation. When a trace is generated for the application model and PIM operations are allocated, they are converted into instructions and passed to both the software and hardware models. The PIM Library enters a waiting state upon receiving tasks and, during this time, performs the interpreted operations in both the software model and hardware. In order to synchronize the timing between hardware and software, the next instruction is issued once both the software and hardware models are in a ready state. Through this process, PIMCoSim ensures that data are shared and synchronized between the software and hardware, allowing for the stable simulation of memory system behavior and PE hardware using the application model. Additionally, the Runtime Library evaluates the overall simulation performance by receiving parameters for the operation of the PIM Library and the computation time of the PE hardware.

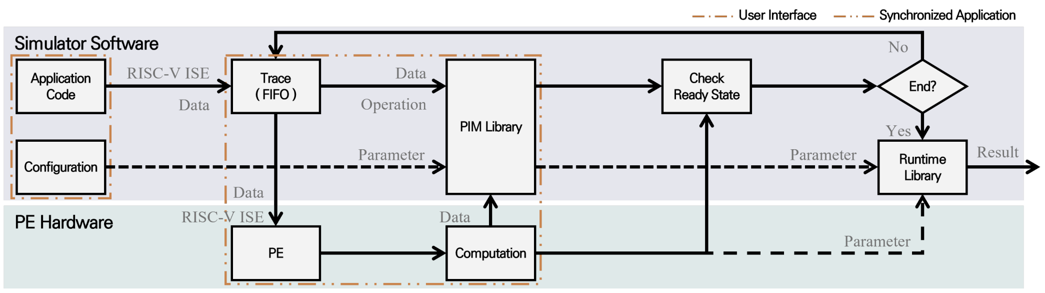


Figure 4. Flow between simulator software and PE hardware.

3.3. Simulator Software

Figure 5 depicts the overall architecture of the software simulator responsible for memory system operations. The simulator receives application code from the researcher, converts it into RISC-V instructions, and transmits it to the hardware via the PIM Library, which models PIM functionality, and the serial peripheral interface (SPI). The application code is written using APIs that allocate PIM operations, and the simulation environment is defined based on the configuration file.

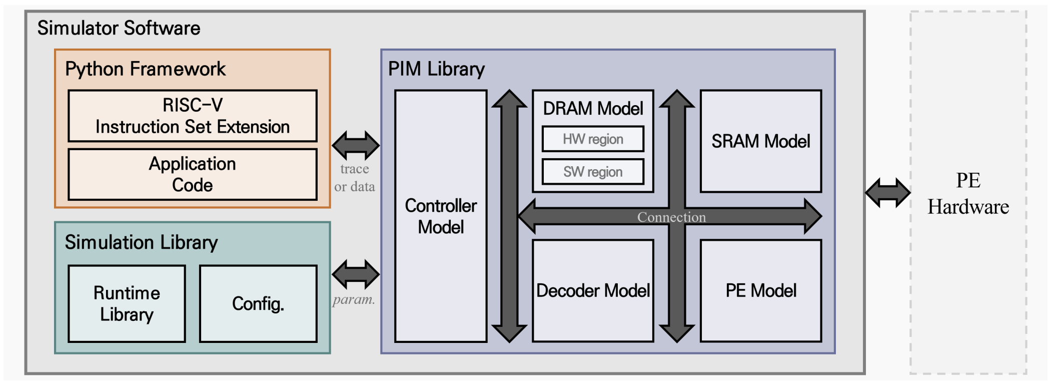


Figure 5. Overview of architecture of the simulator software.

3.3.1. Configurations

The Simulation Library parses the researcher-defined configuration file to model the memory system environment and establish the PIM architecture. Parameters specified in the configuration file are summarized in Table 2. The DRAM structure is designed with parameters that are ranks per channel, banks per rank, burst length, bus width, and PEs per bank. Timing parameters, based on DRAM operations, are integrated into the Runtime Library to support DRAM behavior modeling. Additionally, the SRAM cache is modeled based on timing parameters and the size allocated per PE.

Table 2. PIMCoSim configuration file setting parameters.

Configuration Parameters	
Ranks	The number of ranks per channel, which are independently activated units within the memory. Multiple ranks support more parallel operations.
Banks	The number of banks within each rank, which are units that store data in segments and enable parallel access. More banks improve memory access efficiency.
Burst Length	The amount of data transferred in a single memory access, determining the data transferred per clock cycle.
Bus Width	The width of the memory bus, indicating the amount of data transferred at once between memory and CPU or PIM.
SRAM Cache Size	The size of the SRAM cache associated with each PE, allowing for temporary data storage close to the PE to reduce memory access latency and improve computational efficiency.
PEs per Bank	The number of PEs allocated to each bank, which enhances PIM computational performance through parallel processing.
Timings	Various timing parameters that determine system latency.

3.3.2. RISC-V ISE for PIM Workload

To efficiently operate PIM and support the extension of various instructions, an RISC-V-based instruction set extension (ISE) is provided. RISC-V, an open-source instruction set architecture (ISA) based on RISC principles, offers a simple and efficient base instruction set with the flexibility to add a variety of extended instructions, making it widely used in research.

PIM-specific instructions are defined using the R, I, and S types provided in the RV32I, as shown in Tables 3 and 4. To support memory access in DRAM and PEs, the sw.pim and lw.pim instructions are extended, enabling data movement within specific banks. Instructions are provided for integer and floating-point operations, logical operations, and MAC operations. Since operand sharing and movement among PEs within PIM are critical, additional instructions are available to support these functions. Researchers are able to model the computational requirements of applications by utilizing these provided instructions.

3.3.3. Application Code

PIMCoSim provides an API for developing target applications. The PIMCoSim API converts specified operations in application code into PIM ISE instructions, transforming them into low-level commands and traces that the PIM Library and PE hardware are able to interpret and execute. The converted binary instructions are added to the trace, preserving the execution order and sequentially passing them to the PIM Library model and PE hardware, ensuring that the PIMCoSim is synchronized and executed at the expected time. Therefore, the PIM API not only translates high-level instructions into low-level commands executable on PIM hardware but also serves as an interface for hardware control and the simulation model.

Table 3. RISC-V base instruction format with R, I, and S types.

	31 25	24 20	19 15	14 12	11 7	6 0
R-Type	funct7	rs2	rs1	funct3	rd	opcode
I-Type	imm12		rs1	funct3	rd	opcode
S-Type	imm7	rs2	rs1	funct3	imm5	opcode

Table 4. PIM-specific instructions on PIMCoSim.

Instruction Type	Instruction	Description	
S-Type	`sw.pim`	Stores data from DRAM to PEs.	PE[imm][SRAM[rs1]] = DRAM[rs2]
I-Type	`lw.pim`	Loads data from PEs to DRAM.	DRAM[rd] = PE[imm][SRAM[rs1]]
R-Type	`fadd.pim`	Performs floating-point addition.	PE[funct][SRAM[rd]] = SRAM[rs1] + SRAM[rs2]
	`fsub.pim`	Performs floating-point subtraction.	PE[funct][SRAM[rd]] = SRAM[rs1] − SRAM[rs2]
	`fmul.pim`	Performs floating-point multiplication.	PE[funct][SRAM[rd]] = SRAM[rs1] × SRAM[rs2]
	`iadd.pim`	Performs integer addition.	PE[funct][SRAM[rd]] = SRAM[rs1] + SRAM[rs2]
	`isub.pim`	Performs integer subtraction.	PE[funct][SRAM[rd]] = SRAM[rs1] − SRAM[rs2]
	`imul.pim`	Performs integer multiplication.	PE[funct][SRAM[rd]] = SRAM[rs1] × SRAM[rs2]
	`and.pim`	Performs bitwise AND operation.	PE[funct][SRAM[rd]] = SRAM[rs1] & SRAM[rs2]
	`or.pim`	Performs bitwise OR operation.	PE[funct][SRAM[rd]] = SRAM[rs1] | SRAM[rs2]
	`xor.pim`	Performs bitwise XOR operation.	PE[funct][SRAM[rd]] = SRAM[rs1] ∧ SRAM[rs2]
	`acc.pim`	Accumulates values within range.	PE[funct][SRAM[rd]] = SUM{SRAM[rs1:rs2]}
	`cp.pim`	Copies data from source PE to target PE.	PE[funct][SRAM[rd]] = PE[rs2][SRAM[rs1]]

This enables researchers to effectively utilize complex functionalities of PIM architecture with high-level instructions, simplifying application implementation by removing the need to work directly with hardware interfaces. As a result, researchers are able to easily model computational and memory access policies using application code for implementation.

3.3.4. PIM Modeling

The Controller model manages the operation of other models within the PIM Library and sequentially loads PIM-specific instructions that have been converted into traces and allocates data and computations to the relevant PIM Library models needed for executing the instructions. Simultaneously, the corresponding instructions and necessary data are transmitted to the hardware PE through the SPI. Since the DRAM model in the PIM Library acts as virtual memory within PIMCoSim, data dependency issues arise between the software model and hardware PE when executing memory access instructions. To prevent this, when directly accessing DRAM, the simulation of the PIM Library model is prioritized before passing instructions to the PE hardware. Conversely, when computations are performed within the PE, instructions are first sent to the PE hardware, allowing the software simulator to independently simulate computations while hardware operations are executed. This approach resolves synchronization issues and ensures both efficiency and accuracy by parallelizing co-simulation as needed. Once the synchronization signal from the PE hardware is validated and the software model's computation is complete, the next instruction is loaded from the trace to continue the simulation.

The DRAM model stores data in the IEEE 754 floating-point format, which is used for simulation and computations by the hardware PE. Additionally, by handling the software results and PE hardware computations separately as software DRAM and hardware DRAM, the system enables comparison and debugging of both computation results. A simplified FSM, as depicted in Figure 6, is defined, consisting of states for read, write, activate, precharge, and refresh operations to model the DRAM. Even when DRAM is continuously accessed in the simulation, unused cells still require refreshing. Therefore, a wait time equal to the refresh cycle time (tRFC) is applied at each refresh interval (tREFI). In the case of a read operation, the process transitions from the IDLE state to the active state to activate the row, then waits for the row-to-column delay (tRCD) time to read the data. Subsequently, during a read command, timing is modeled with column address strobe (CAS) latency (tCL) and row active time (tRAS) to ensure timing accuracy and data stability. Once the read operation is completed, the FSM performs a precharge operation during the row precharge time (tRP) and transitions back to the IDLE state. For a write operation, the transition from the IDLE state to the active state is the same as in the read process. Then, during the write command, there is a wait for the CAS write latency (tCWL) before data are written.

After the write operation is completed, the write recovery time (tWR) allows for stable data storage, followed by closing the row for the tRP duration, transitioning back to the IDLE state. The cycle counts for each step are parameterized and passed to the Runtime Library to model DRAM functionality.

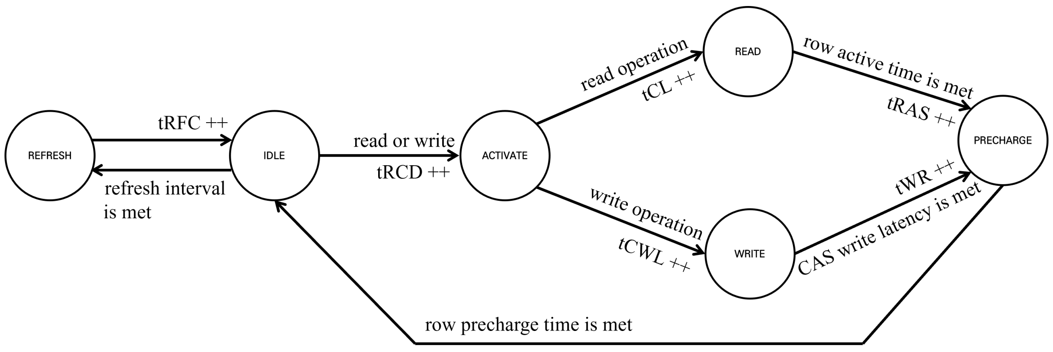


Figure 6. State diagram for DRAM modeling.

The SRAM cache model receives data needed for PE computations from the DRAM model and facilitates storage and exchange. Each PE model is assigned an SRAM cache model, where data are stored based on the address interpreted from instructions. Following the same synchronization process used for DRAM data, the actual SRAM in PE hardware and the software SRAM cache model store identical data. The SRAM cache is modeled in a simplified form using read and write latency parameters. According to [42], the write-back policy was adopted to optimize performance, as write-through cache structures reportedly cause a performance degradation.

A specified number of PE models are allocated to each bank, serving as computation kernels and performing operations according to PIM-specific instructions using data stored in the SRAM cache. Based on the given instructions, they execute addition, multiplication, and MAC operations, and exchange integer and floating-point computation results with the SRAM cache. The use of the write-back policy allows the PE models to perform repeated operations on cached data efficiently, minimizing DRAM access latency and improving overall system performance.

3.4. PE Hardware

Figure 7 shows the overall architecture of the PE implemented in hardware. It consists of a data controller module to control data path within the PE hardware, a module for serial communication with the simulator software, a module for data exchange with the SRAM, and an operator module containing the floating-point unit (FPU) where actual computations are performed.

The data controller module includes a decoder to interpret instructions received from the simulator software and an FSM to perform operations for each instruction. The serial communication module and the SRAM control module include interfaces to match the bit length of 1-word between connections. The operator module comprises a 32-bit FPU capable of addition, subtraction, multiplication, and absolute operations, and an accumulator consisting of four floating-point adders to perform accumulation operations within four clock cycles.

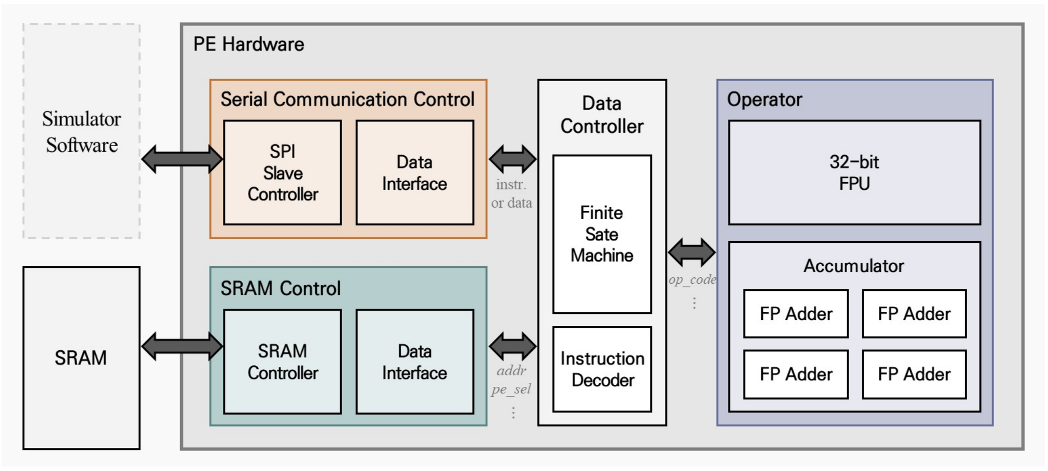


Figure 7. Overview of architecture of the PE hardware.

3.4.1. Data Controller

To perform operations for each instruction within the PE hardware, the data controller module first decodes the instruction upon receipt. The decoder interprets the 32-bit custom instruction based on RISC-V, received from the simulator software, and outputs signals indicating the operation to be performed (*op_code*), the address of the SRAM for reading/writing data (*addr*), and the PE within the virtual PE array where the operation is to be executed (*pe_sel*).

Figure 8 illustrates the state diagram for the FSM included in the data controller module. When a 32-bit signal is received from the simulator software in the *IDLE* state, it determines whether the signal is an instruction or data. For write operations to the SRAM, a data signal always follows the instruction signal, so the FSM validates if it is ordered to receive the data signal after interpreting the instruction. If it is determined to be an instruction, the state transitions to *DEC* (decode), and if it is data, the state transitions to *W_EX_DATA* (write external data). In the *W_EX_DATA* state, since it involves writing data received from external sources to the SRAM, the data are repeatedly written to the address regions of the SRAM corresponding to all selected PEs. After interpreting the instruction in the *DEC* state and determining the operation to be executed, the state transitions accordingly. Except for the *RDATA1* (read 1 data) and *CP_RDATA* (read and copy data) states, which read SRAM data for a single PE, all other states repeat the operation until the signal *pe_last*, which indicates if the operation is repeated for the number of selected PEs, becomes '1'. The abbreviations for the states not mentioned are as follows:

• *RDATA2*	:	read 2 data.
• *RDATA9*	:	read 9 data.
• *TXDATA*	:	transmit data to external.
• *OP*	:	operate.
• *ACC*	:	accumulate.
• *W_CP_DATA*	:	write copied data.
• *W_OP_RES*	:	write operational result.
• *W_ACC_RES*	:	write accumulation result.

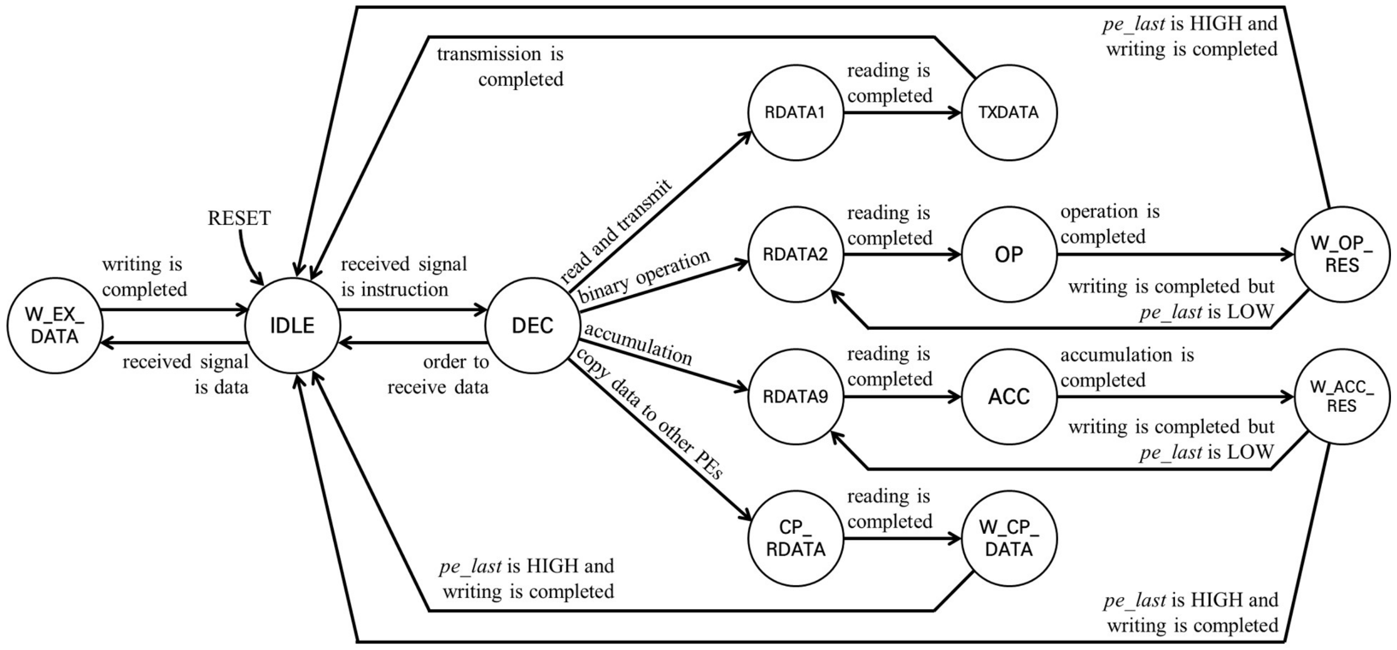


Figure 8. State diagram for FSM of the PE hardware.

3.4.2. Communication with Simulator Software

Signals such as instructions and data are exchanged between the PE hardware and the simulator software via the SPI communication. The SPI mode employed in this PIMCoSim is mode 3 ($CPHA = 1$, $CPOL = 1$), and each communication transmits and receives 8-bit signals. However, since 1-word in the PIMCoSim system is 32-bit, an interface module is required to conduct communication four times to create a 32-bit signal.

When communicating between the simulator software (master) and the PE hardware (slave), it is crucial to perform the communication with proper timing. If the software sends a signal to communicate when the hardware is not ready, unintended errors occur. Therefore, in the proposed PIMCoSim, an *spi_irq* signal is added to notify the software when the PE hardware needs to communicate. Specifically, the FSM in the data controller module indicates that the communication is possible when it is in the *IDLE* state, and the simulator software waits until the signal is '1' before beginning the communication.

3.4.3. SRAM Controller

The SRAM utilized in the PIMCoSim is ISSI IS61WV102416ALL, which supports fully static operation, eliminating the need for clock or refresh cycles. Each word in the SRAM is 16-bit, and the address is 20-bit. The SRAM controller module consists of a submodule that generates signals matching the SRAM read/write cycle waveform and a submodule that controls the bidirectional pins for data exchange between the controller and the SRAM.

Similar to SPI communication, an interface is required to match the word size with the PIMCoSim system. Since the word size of the SRAM controller is 16-bit, data are exchanged by combining two data units and performing burst read/write operations.

3.4.4. Operator

The operator module consists of an FPU for performing 32-bit floating-point- or integer-based PIM operations and an accumulator for accumulation in the MAC operations required for convolutions. The PIMCoSim utilizes up to nine PEs per instruction, enabling the addition of nine numbers during accumulation operations. Since the floating-point adder applied in the PIMCoSim only adds two numbers at a time, the maximum number of parallel addition operations for nine numbers is four, requiring four sequential stages of operations.

Figure 9 shows the architectural diagram of the accumulator module. As explained above, the maximum number of parallel addition operations is four, so the module includes a total of four floating-point adders. A controller included in the accumulator module ensures that the results of the addition operations are fed back into the floating-point adders. This design allows the addition of nine numbers to be completed in a total of four clock cycles.

The designed floating-point unit (FPU) comprises a floating-point adder and multiplier for conducting 32-bit real number operations, and an ALU for integer operations. Both the adder and multiplier adopt the IEEE 754 standard for floating-point arithmetic. Additionally, a decoder is integrated to interpret operation codes and select the appropriate module among the adder, multiplier, or ALU. The FPU is capable of performing various operations including addition, subtraction, and multiplication for real numbers, as well as addition, subtraction, multiplication, and logical operations (AND, OR, XOR) for integers. Based on the input of two numbers and an operand code, the FPU executes the specified operation and outputs the result after a two-clock-cycle delay.

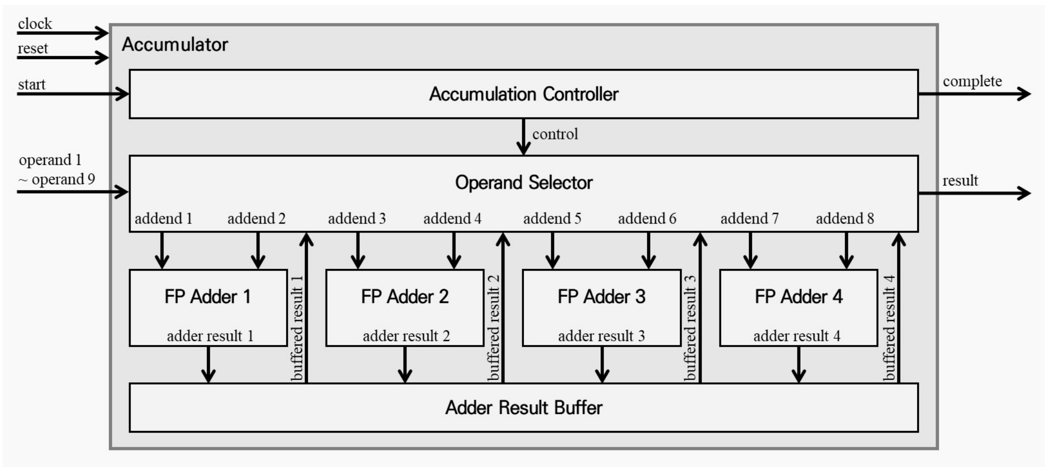


Figure 9. An architectural diagram of the accumulator.

4. Implementation and Evaluation

This section evaluates the efficiency of a DRAM-PIM system by optimizing application benchmarks and the number of PEs in the PIMCoSim.The PIMCoSim was implemented as a Python program on a Raspberry Pi 4, while the PE hardware was implemented on an Intel FPGA (Cyclone IV EP4CE115F29C7N) with 2MB SRAM (IS61WV102416BLL) and synthesized by Quartus, targeting an operating frequency of 50 MHz. To verify the feasibility of the PIMCoSim, we also developed a software simulator that performs operations of the PIM-specific instructions.

4.1. Experimental Use Case

Table 5 presents benchmarks for general matrix–vector multiplication (GEMV) and general matrix–matrix multiplication (GEMM) kernels applied to the CNN, and Transformer models. GEMV and GEMM are utilized in fully connected layers or convolutional layers, and matrix sizes employed in various models were selected for benchmarking. The GEMV is expressed as $y = \alpha Ax + \beta y$, where A and B are matrix inputs, α and β are scalar inputs, and x and y are the vector input and vector output, which is overwritten by the output, respectively. The GEMM is expressed as $C = \alpha AB + \beta C$, where C is the pre-existing matrix, which is overwritten by the output.

Table 5. Benchmark.

Case	GEMV Dimension	Case	GEMM Dimension
GEMV1	256 × 1k		
GEMV2	512 × 1k	GEMM1	256 × 1k × 9
GEMV3	512 × 2k		
GEMV4	1k × 2k		
GEMV5	1k × 4k	GEMM2	512 × 1k × 9
GEMV6	2k × 4k		

The baseline DRAM-CPU system references gem5, focusing on memory system simulation through the system-call emulation mode. The gem5 configuration employs a 2.90 GHz RISC-V CPU to measure memory accesses. Table 6 shows the memory system configurations for gem5 and PIMCoSim, with the parameters.

Table 6. Specification of PIMCoSim system.

Memory Configuration	
Timing	tCL = tRCD = tRP = tCWL = 13.75 ns, tRAS = 35 ns, tWR = 15 ns, tRFC = 260 ns, tREFI = 7800 ns
DRAM	1 channel, 2 ranks/channel, 8 banks/rank, 8-burst length, 8B bus width
PIM Configuration	
PE	1 FPU/PE, 1 accumulator/PE, (1, 3, 9) PEs/bank
SRAM Cache	128B/PE

To simulate GEMV and GEMM in gem5, applications were written in C and compiled to the RV32I format by the RISC-V GNU toolchain 13.2.0. In contrast, application code for GEMV and GEMM in PIMCoSim was written in Python, with variations in the number of PEs located in each bank for each application. To optimize parallel processing of matrix operations, the Fox algorithm was adopted, allowing multiple PEs in each bank to perform partial matrix operations simultaneously and aggregate the results. The experiment evaluated performance differences based on memory accesses for each application and the number of PEs per PIM bank, comparing the impact of PIM architecture and computation policies on matrix computation performance.

4.2. Experimental Results

We experimentally evaluated the CPU and memory access patterns for GEMV and GEMM operations across various matrix sizes to assess the efficiency of PIM operations.

First, we compared memory accesses for each benchmark between the traditional CPU-based memory system and the DRAM-PIM memory system, examining both the total accesses and the proportion of each access type based on the number of PEs allocated per PIM bank, as shown in Figure 10. Because the goal of PIM is to reduce memory bottlenecks by decreasing memory accesses, these factors were tested in the experiment. In PIM-DRAM, the proportion of direct accesses to DRAM is lower than in CPU-based memory systems, while overall cache access patterns tend to remain consistent regardless of the increase in the number of PEs. This is attributed to the observation that, as the number of PEs increases in the same application, the number of operands written back to the cache decreases, while the number of operands loaded or copied remains constant.

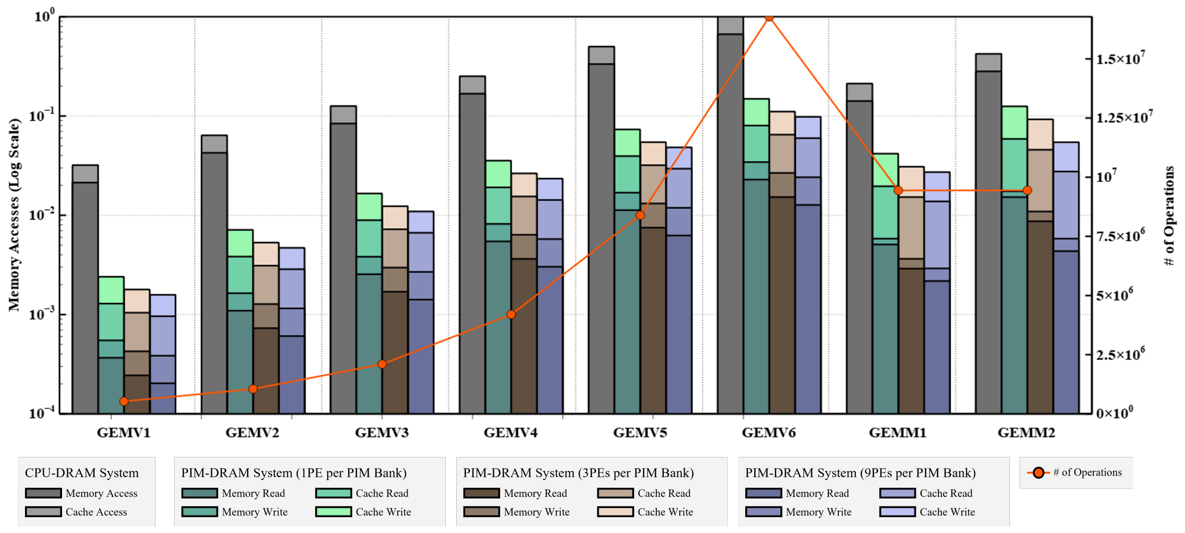


Figure 10. Impact of the number of PEs and processing optimization on memory access patterns.

Second, we compared the execution times of emulated PE hardware for each benchmark, as shown in Figure 11. In the case of GEMV, the computation time showed an increasing trend as the dimensions grew. However, as the number of PEs increased, a significant reduction in computation time was observed. Compared to a single PE, three PEs resulted in a speed improvement ranging from 1.7 to 1.9 times, while nine PEs achieved a performance increase between 2.4 and 3.5 times. The performance degradation with increasing GEMM dimensions was more pronounced than with GEMV since GEMM operations are more computationally intensive than GEMV. Using three PEs led to a performance improvement of about 1.2 times compared to a single PE, while nine PEs resulted in an improvement of approximately 2.2 times. While the GEMV application demonstrated effective performance improvements, GEMM showed limited gains even with an increased number of PEs. This suggests that GEMM requires optimization through tailored computation policies or a further increase in PEs for additional parallel processing.

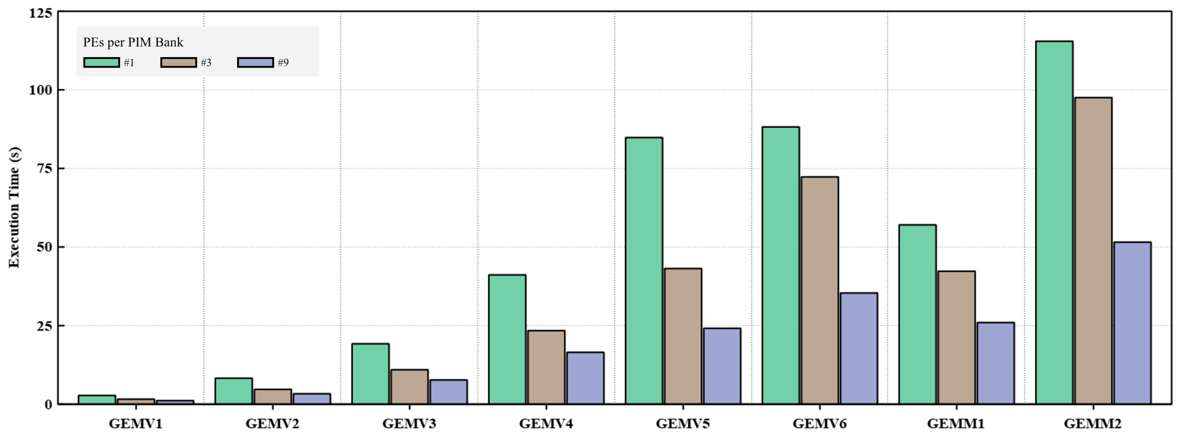


Figure 11. Impact of the number of PEs and optimization on the execution time of PE hardware.

Thirdly, we evaluated the computational accuracy and performance by comparing the mean squared error (MSE) results among the host, software simulator, and emulated PE hardware for each benchmark, as shown in Table 7. The MSE values in GEMV and GEMM showed very minor differences, confirming the accuracy of the default PE hardware computation results and the consistency of the software simulator. This experiment provides reliable results for performance evaluation across various applications and verifies that the computation performance of PE hardware is effectively analyzed and debugged.

Table 7. MSE analysis of computation performance.

Case	PEs per PIM Bank	GEMV ($\times 10^{-6}$)						GEMM ($\times 10^{-9}$)	
		GEMV1	GEMV2	GEMV3	GEMV4	GEMV5	GEMV6	GEMM1	GEMM2
Host vs. Simulator	1	0.631	0.617	0.61	0.61	0.366	0.306	0.113	0.102
	3	0.658	0.64	0.559	0.628	0.597	0.692	0.057	0.082
	9	0.598	0.558	0.506	0.5	0.623	0.672	0.012	0.064
Simulator vs. PE	1	0.094	0.105	0.26	0.251	0.884	1.02	0.177	0.194
	3	0.055	0.055	0.172	0.127	0.446	0.43	0.254	0.255
	9	0.031	0.041	0.104	0.077	0.239	0.264	0.263	0.264
Host vs. PE	1	0.725	0.722	0.87	0.861	1.25	1.326	0.29	0.296
	3	0.713	0.695	0.731	0.755	1.043	1.122	0.311	0.337
	9	0.629	0.599	0.61	0.577	0.852	0.936	0.275	0.328

We confirmed that the DRAM-PIM memory system shows improvements in memory accesses and computational performance compared to a CPU-based memory system in GEMV and GEMM operations. Notably, increasing the number of PEs led to greater performance gains in GEMV, due to the data access pattern aligning effectively with the PIM architecture. In contrast, GEMM showed relatively limited performance gains, indicating the need for further optimization to address the computational complexity. These findings highlight both the strengths and limitations of the PIM architecture, underscoring the importance of optimizing PIM for various computational environments.

PIMCoSim successfully evaluates the efficiency of PIM operations, enabling the analysis of system behavior. By partially emulating simplified PEs without emulating the entire system, it allows for effective examination of hardware characteristics and performance assessment in a flexible environment. Specifically, by utilizing the PIM-specific instruction set provided by the simulator to simulate GEMV and GEMM operations across various matrix sizes, PIMCoSim demonstrates the potential by showing concrete improvements in memory access patterns and computational performance compared to a CPU-based memory system.

5. Conclusions

This paper introduces PIMCoSim, a comprehensive co-simulation framework designed to optimize and evaluate DRAM-PIM architectures. PIMCoSim provides a flexible simulation environment that integrates both hardware and software models, enabling precise evaluation of PIM systems for machine learning and other data-intensive applications. The cycle-accurate emulation enables researchers to analyze detailed memory access patterns and optimize PIM-specific instructions within various DRAM configurations. Additionally, the PIM-specific instruction set allows applications to be implemented within the software simulation, while supporting RISC-V ISE to enable researchers to extend instructions suitable for target workloads and apply them to DRAM-PIM systems. For PIM DSE, researchers modify the characteristics of PEs located in banks, enabling access to various PE computation structures and facilitating comparisons of access policies using PIM-specific instructions.

In order to verify the validity of PIMCoSim, we implemented the co-simulator on a Raspberry Pi 4 and an Intel FPGA. We compared the memory access counts for each benchmark and the execution times of the emulated PE hardware between CPU-based memory systems and DRAM-PIM memory systems. The experimental results demonstrated significant improvements in performance and efficiency by leveraging PIM operations to reduce data movement bottlenecks commonly encountered in conventional CPU-based systems.

Contribution of the PIMCoSim is centered on enabling cross-layer optimization by effectively aligning hardware and software tasks to maximize the potential of PIM. Additionally, PIMCoSim provides the flexibility to examine hardware characteristics and assess performance without emulating the entire system, by utilizing PE hardware implemented on hardware along with a modeled software memory system. Future research will focus on extending the functionality of the PIMCoSim to provide more configurable parameters, exploring additional optimizations, and evaluating DRAM-PIM systems across various computing environments.

The PIMCoSim in this work lacks the capability to thoroughly analyze the impact of various bit widths on system performance or systematically compare the performance differences of cache policies. To address these limitations, future work aims to enhance the co-simulator to enable the design of instruction architectures optimized for specific workloads and to analyze performance variations based on data consistency and memory access costs. These improvements are expected to support the development of more efficient PIM architecture designs.

Author Contributions: Conceptualization, J.S., S.A., S.L. and S.E.L.; methodology, J.S., S.A., S.L. and S.E.L.; software, J.S.; hardware, S.A. and S.L.; validation, J.S., S.A., S.L. and S.E.L.; data curation, J.S., S.A. and S.L.; investigation, J.S. and S.A.; writing—original draft preparation, J.S. and S.A.; writing—review and editing, J.S., S.A., S.L. and S.E.L.; supervision, S.E.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2024-RS-2022-00156295) supervised by the IITP (Institute of Information & Communications Technology Planning & Evaluation). This work was partly supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (IITP-2022-0-01013, Development of DRAM PIM semiconductor technology for enhanced computing function for edge).

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AI	artificial intelligence
GPU	graphics processing unit
FPGA	field-programmable gate array
ASIC	application-specific integrated circuit
DRAM	dynamic random-access memory
ReRAM	resistive random-access memory
ALU	arithmetic and logical unit
PIM	processing in memory
PE	processing element
DSE	design space exploration
MVM	matrix–vector multiplication
MAC	multiply–accumulate
HBM	high-bandwidth memory
FPU	floating-point unit
HMC	hybrid memory cube
TSV	Through-Silicon Vias
CNN	convolutional neural network
SRAM	static random-access memory
SPI	serial peripheral interface
ISA	instruction set architecture
ISE	instruction set extension
CAS	column address strobe
FSM	finite state machine
MSE	mean squared error
GEMV	general matrix–vector multiplication
GEMM	general matrix–matrix multiplication

References

1. Park, J.; Shin, J.; Kim, R.; An, S.; Lee, S.; Kim, J.; Oh, J.; Jeong, Y.; Kim, S.; Jeong, Y.R.; et al. Accelerating Strawberry Ripeness Classification Using a Convolution-Based Feature Extractor along with an Edge AI Processor. *Electronics* **2024**, *13*, 344. [CrossRef]
2. Guo, X.; Wang, J.; Gao, G.; Li, L.; Zhou, J.; Li, Y. Improving Text Classification in Agricultural Expert Systems with a Bidirectional Encoder Recurrent Convolutional Neural Network. *Electronics* **2024**, *13*, 4054. [CrossRef]
3. Seng, K.P.; Ang, L.-M.; Peter, E.; Mmonyi, A. Machine Learning and AI Technologies for Smart Wearables. *Electronics* **2023**, *12*, 1509. [CrossRef]
4. Gholami, A.; Yao, Z.; Kim, S.; Hooper, C.; Mahoney, M.W.; Keutzer, K. AI and Memory Wall. *IEEE Micro* **2024**, *44*, 33–39. [CrossRef]
5. Kim, J.; Kim, R.; Oh, J.; Lee, S.E. Hardware-Based WebAssembly Accelerator for Embedded System. *Electronics* **2024**, *13*, 3979. [CrossRef]
6. Lee, S.; An, S.; Kim, J.; Namkung, H.; Park, J.; Kim, R.; Lee, S.E. Grid-Based DBSCAN Clustering Accelerator for LiDAR's Point Cloud. *Electronics* **2024**, *13*, 3395. [CrossRef]
7. Keckler, S.W.; Dally, W.J.; Khailany, B.; Garland, M.; Glasco, D. GPUs and the Future of Parallel Computing. *IEEE Micro* **2011**, *31*, 7–17. [CrossRef]
8. An, S.; Oh, J.; Lee, S.; Kim, J.; Jeong, Y.; Kim, J.; Lee, S.E. Lightweight and Error-Tolerant Stereo Matching with a Stochastic Computing Processor. *Electronics* **2024**, *13*, 2024. [CrossRef]
9. Seshadri, V.; Kim, Y.; Fallin, C.; Lee, D.; Ausavarungnirun, R.; Pekhimenko, G.; Luo, Y.; Mutlu, O.; Gibbons, P.B.; Kozuch, M.A.; et al. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46), Davis, CA, USA, 7–11 December 2013 ; Association for Computing Machinery: New York, NY, USA, 2013; 185–197. [CrossRef]
10. Chen, Y. ReRAM: History, Status, and Future. *IEEE Trans. Electron Devices* **2020**, *67*, 1420–1433. [CrossRef]
11. Asifuzzaman, K.; Miniskar, N.R.; Young, A.R.; Liu, F.; Vetter, J.S. A survey on processing-in-memory techniques: Advances and challenges. *Mem. Mater. Devices, Circuits Syst.* **2023**, *4*, 100022. [CrossRef]
12. Lim, J.; Son, J.; Yoo, H. Efficient Processing-in-Memory System Based on RISC-V Instruction Set Architecture. *Electronics* **2024**, *13*, 2971. [CrossRef]
13. Kaur, R.; Asad, A.; Mohammadi, F. A Comprehensive Review of Processing-in-Memory Architectures for Deep Neural Networks. *Electronics* **2024**, *13*, 174. [CrossRef]
14. Han, C.; Jeong, Y.; Lee, S.E. Simulation-Based Fault Analysis for Resilient System-On-Chip Design. *J. Inf. Commun. Converg. Eng.* **2021**, *19*, 175. [CrossRef]
15. Hwang, D.H.; Han, C.Y.; Oh, H.W.; Lee, S.E. ASimOV: A Framework for Simulation and Optimization of an Embedded AI Accelerator. *Micromachines* **2021**, *12*, 838. [CrossRef] [PubMed]
16. Gabbay, F.; Lev Aharoni, R.; Schweitzer, O. Deep Neural Network Memory Performance and Throughput Modeling and Simulation Framework. *Mathematics* **2022**, *10*, 4144. [CrossRef]
17. Biancolin, D.; Karandikar, S.; Kim, D.; Koenig, J.; Waterman, A.; Bachrach, J.; Asanovic, K. FASED: FPGA-Accelerated Simulation and Evaluation of DRAM. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), Seaside, CA, USA, 24–26 February 2019 ; Association for Computing Machinery: New York, NY, USA, 2019; pp. 330–339. [CrossRef]
18. Heo, J.; Shin, Y.; Choi, S.; Yune, S.; Kim, J.H.; Sung, H.; Kwon, Y.; Kim, J.Y. PRIMO: A Full-Stack Processing-in-DRAM Emulation Framework for Machine Learning Workloads. In Proceedings of the 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD), San Francisco, CA, USA, 29 October–2 November 2023; pp. 1–9. [CrossRef]
19. Krammer, M.; Schiffer, C.; Benedikt, M. ProMECoS: A Process Model for Efficient Standard-Driven Distributed Co-Simulation. *Electronics* **2021**, *10*, 633. [CrossRef]
20. Barukčić, M.; Varga, T.; Jerković Štil, V.; Benšić, T. Co-Simulation Framework for Optimal Allocation and Power Management of DGs in Power Distribution Networks Based on Computational Intelligence Techniques. *Electronics* **2021**, *10*, 1648. [CrossRef]
21. Biagetti, G.; Falaschetti, L.; Crippa, P.; Alessandrini, M.; Turchetti, C. Open-Source HW/SW Co-Simulation Using QEMU and GHDL for VHDL-Based SoC Design. *Electronics* **2023**, *12*, 3986. [CrossRef]
22. Chen, W.; Qi, Z.; Akhtar, Z.; Siddique, K. Resistive-RAM-Based In-Memory Computing for Neural Network: A Review. *Electronics* **2022**, *11*, 3667. [CrossRef]
23. Long, Y.; Na, T.; Mukhopadhyay, S. ReRAM-Based Processing-in-Memory Architecture for Recurrent Neural Network Acceleration. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2018**, *26*, 2781–2794. [CrossRef]
24. Jin, H.; Liu, C.; Liu, H.; Luo, R.; Xu, J.; Mao, F.; Liao, X. ReHy: A ReRAM-Based Digital/Analog Hybrid PIM Architecture for Accelerating CNN Training. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 2872–2884. [CrossRef]
25. Roy, S.; Ali, M.; Raghunathan, A. PIM-DRAM: Accelerating Machine Learning Workloads Using Processing in Commodity DRAM. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2021**, *11*, 701–710. [CrossRef]
26. He, M.; Song, C.; Kim, I.; Jeong, C.; Kim, S.; Park, I.; Thottethodi, M.; Vijaykumar, T.N. Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning. In Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, 17–21 October 2020; pp. 372–385. [CrossRef]

27. Lee, S.; Kang, S.H.; Lee, J.; Kim, H.; Lee, E.; Seo, S.; Yoon, H.; Lee, S.; Lim, K.; Shin, H.; et al. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product. In Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 14–18 June 2021; pp. 43–56. [CrossRef]
28. Nai, L.; Hadidi, R.; Sim, J.; Kim, H.; Kumar, P.; Kim, H. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), Austin, TX, USA, 4–8 February 2017; pp. 457–468. [CrossRef]
29. Lockerman, E.; Feldmann, A.; Bakhshalipour, M.; Stanescu, A.; Gupta, S.; Sanchez, D.; Beckmann, N. Livia: Data-Centric Computing Throughout the Memory Hierarchy. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), Lausanne, Switzerland, 16–20 March 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 417–433. [CrossRef]
30. Li, S.; Niu, D.; Malladi, K.T.; Zheng, H.; Brennan, B.; Xie, Y. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17), Cambridge, MA, USA, 14–18 October 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 288–301. [CrossRef]
31. Xu, S.; Chen, X.; Wang, Y.; Han, Y.; Qian, X.; Li, X. PIMSim: A Flexible and Detailed Processing-in-Memory Simulator. *IEEE Comput. Archit. Lett.* **2019**, *18*, 6–9. [CrossRef]
32. Rosenfeld, P.; Cooper-Balis, E.; Jacob, B. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Comput. Archit. Lett.* **2011**, *10*, 16–19. [CrossRef]
33. Leidel, J.D.; Chen, Y. HMC-SIM: A Simulation Framework for Hybrid Memory Cube Devices. *Parallel Process. Lett.* **2014**, *24*, 1442002. [CrossRef]
34. Poremba, M.; Xie, Y. NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI, Amherst, MA, USA, 19–21 August 2012; pp. 392–397. [CrossRef]
35. Binkert, N.; Beckmann, B.; Black, G.; Reinhardt, S.K.; Saidi, A.; Basu, A.; Hestness, J.; Hower, D.R.; Krishna, T.; Sardashti, S.; et al. The gem5 simulator. *ACM SIGARCH Comput. Archit. News* **2011**, *39*, 1–7. [CrossRef]
36. Santos, P.C.; Forlin, B.E.; Carro, L. Sim2PIM: A Fast Method for Simulating Host Independent & PIM Agnostic Designs. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 1–5 February 2021; pp. 226–231. [CrossRef]
37. Yu, C.; Liu, S.; Khan, S. MultiPIM: A Detailed and Configurable Multi-Stack Processing-In-Memory Simulator. *IEEE Comput. Archit. Lett.* **2021**, *20*, 54–57. [CrossRef]
38. Oliveira, G.F.; Santos, P.C.; Alves, M.A.Z.; Carro, L. A generic processing in memory cycle accurate simulator under hybrid memory cube architecture. In Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), Pythagorion, Greece, 17–20 July 2017; pp. 54–61. [CrossRef]
39. Zhang, J.; Zha, Y.; Beckwith, N.; Liu, B.; Li, J. MEG: A RISCV-based System Emulation Infrastructure for Near-data Processing Using FPGAs and High-bandwidth Memory. *ACM Trans. Reconfigurable Technol. Syst.* **2020**, *13*, 1–24. [CrossRef]
40. Mosanu, S.; Sakib, M.N.; Tracy, T.; Cukurtas, E.; Ahmed, A.; Ivanov, P.; Khan, S.; Skadron, K.; Stan, M. PiMulator: A Fast and Flexible Processing-in-Memory Emulation Platform. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 14–23 March 2022; pp. 1473–1478. [CrossRef]
41. Li, S.; Yang, Z.; Reddy, D.; Srivastava, A.; Jacob, B. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Comput. Archit. Lett.* **2020**, *19*, 106–109. [CrossRef]
42. Shin, Y.; Park, J.; Cho, S.; Sung, H. PIMFlow: Compiler and Runtime Support for CNN Models on Processing-in-Memory DRAM. In Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO'23), Montreal, QC Canada, 25 February–1 March 2023; Association for Computing Machinery: New York, NY, USA, 2023; pp. 249–262. [CrossRef]