Article

Efficient Processing-in-Memory System Based on RISC-V Instruction Set Architecture

Jihwan Lim, Jeonghun Son and Hoyoung Yoo *

Department of Electronics Engineering, Chungnam National University, Daejeon 34134, Republic of Korea; jihwan.lim@abov.co.kr (J.L.); jhsohn.cas@o.cnu.ac.kr (J.S.)
* Correspondence: hyyoo@cnu.ac.kr

Abstract: A lot of research on deep learning and big data has led to efficient methods for processing large volumes of data and research on conserving computing resources. Particularly in domains like the IoT (Internet of Things), where the computing power is constrained, efficiently processing large volumes of data to conserve resources is crucial. The processing-in-memory (PIM) architecture was introduced as a method for efficient large-scale data processing. However, PIM focuses on changes within the memory itself rather than addressing the needs of low-cost solutions such as the IoT. This paper proposes a new approach using the PIM architecture to overcome memory bottlenecks effectively in domains with computing performance constraints. We adopt the RISC-V instruction set architecture for our proposed PIM system's design, implementation, and comprehensive performance evaluation. Our proposal expects to efficiently utilize low-spec systems like the IoT by minimizing core modifications and introducing PIM instructions at the ISA level to enable solutions that leverage PIM capabilities. We evaluate the performance of our proposed architecture by comparing it with existing structures using convolution operations, the fundamental unit of deep-learning and big data computations. The experimental results show our proposed structure achieves a 34.4% improvement in processing speed and 18% improvement in power consumption compared to conventional von Neumann-based architectures. This substantiates its effectiveness at the application level, extending to fields such as deep learning and big data.

Keywords: processing in memory (PIM); artificial intelligence (AI); machine learning; deep learning; RISC-V; Internet of Things (IoT)

check for updates

Citation: Lim, J.; Son, J.; Yoo, H. Efficient Processing-in-Memory System Based on RISC-V Instruction Set Architecture. *Electronics* **2024**, *13*, 2971. https://doi.org/10.3390/electronics13152971

Academic Editor: Sunggu Lee

Received: 17 June 2024
Revised: 22 July 2024
Accepted: 25 July 2024
Published: 27 July 2024

1. Introduction

With the advancements in big data, deep learning, and IoT, research focusing on efficient large-scale data processing and resource conservation has become pivotal in each respective field [1]. However, despite much research, algorithms still consume a lot of computing resources due to structural limitations in how large-scale data are read from memory and processed, and the research of efficient structures remains an important research topic [2]. These structural limitations come from memory bottlenecks typical in von Neumann architectures, where all calculations are processed by the core, necessitating the loading of extensive data from memory [3]. Thus, the speed of memory access becomes a major limiting factor [4]. Yet, due to physical constraints, memory access speeds are significantly lower compared to core processing speeds, thereby making the performance of algorithms processing large-scale data heavily reliant on the speed of loading the data from memory [3].

Research to improve memory access limitations include research on increasing memory bandwidth, enhancing memory speeds, and exploring graphics processing unit (GPU) technologies [5]. High-bandwidth memory (HBM), for instance, proposes the use of high-bandwidth memory interfaces to increase data transfer rates, but structural bottlenecks continue to be important challenges [6]. And GPU uses parallel computing logic to offload computations from the core, acting as accelerators. However, the bottleneck in transferring data from the memory to the GPU complicates achieving optimal performance [7]. To make improvements in the bottlenecks, recent PIM architectures are proposed [8]. Unlike traditional von Neumann architectures where only the core processes and memory stores data, PIM integrates the memory and processing to efficiently perform data processing, offering a promising approach to overcoming memory bottlenecks [9]. However, research on such PIM architectures has primarily focused on optimizing the in-memory computation logic rather than developing PIM solutions for low-cost environments such as the IoT. Consequently, studies on how to effectively utilize PIM architectures within the IoT remain a necessary and unresolved area of research [10].

This paper proposes a novel structure combining PIM architecture with computing resources constrained by a low area and low performance, as an alternative to mitigate the data movement bottleneck between the memory and processors inherent in traditional computer architectures [11]. Specifically, we introduce a PIM system based on the RISC-V instruction set architecture, demonstrating effective strategies to alleviate memory bottlenecks prevalent in existing computing systems [12]. Evaluating our proposed PIM system involves assessing its performance through fundamental operations such as convolution in deep learning [13]. The experimental results highlight the enhanced performance of the proposed architecture in data-intensive fields like deep learning and big data, underscoring its capability to facilitate efficient large-scale data processing in resource-constrained environments such as the IoT [14]. We propose our architecture in the following sequence. In Section 2, Background, we introduce the fundamental information necessary to describe our proposed architecture. In Section 3, Proposed Design, we provide an explanation of our proposed architecture, demonstrate its feasibility, and compare it with existing architectures to highlight improvements. In Section 4, Experimental Results, we evaluate the algorithm and circuit performance of both the proposed architecture and the conventional von Neumann architecture, thereby proving the superiority of our approach. Finally, in Section 5, Conclusion, we summarize our claims and outline future research directions.

2. Background

The background section provides essential information necessary for this study. Firstly, it introduces the concept of convolution operation, which serves as the fundamental computational unit in deep-learning algorithms. Secondly, we describe the RISC-V RV32I architecture, which is based on the traditional von Neumann architecture and utilizes the RISC instruction set.

2.1. Convolution

Convolution is an important operation in deep learning and image processing, used to apply filters to signals or images, create new signals or images, or identify specific features in an image. This operation involves computing the sum of element-wise products between an input signal (or image) and a kernel (filter). The mathematical representation of convolution is shown in Equation (1) [15].

$$[f * g](i,j) = \sum_{d=0}^{n}\sum_{p=0}^{k}\sum_{q=0}^{k} f(p,g,d)\cdot g(p+i,g+j,d) \quad (1)$$

Here, f represents a kernel of size k, g represents an input image, and $*$ denotes the convolution operator. Moving the filter f across the input data g, a single multiply-accumulate operation is performed between corresponding elements. Convolution calculates a weighted sum of neighboring pixel values to generate a new pixel value $[f * g](i,j)$. Figure 1 illustrates the concept of the convolution operation. For an input of size $H_{in} \times W_{in} \times N$, a $k \times k \times n$ kernel performs convolution. It computes the element-wise product between the input data and kernel data at the same location, followed by summation to determine the output data. This process is repeated by shifting the kernel, generating output data of size $H_{out} \times W_{out}$.

Using convolution, various types of filters can be applied to images, enabling operations such as sharpening, edge detection, or blurring. Convolution is also fundamental in deep learning, where convolutional layers are composed of these operations to extract features from images. Through iterative operations, convolution layers learn optimal kernel values. Ultimately, convolutional neural networks (CNNs) utilize these learned convolutional layers to classify images or perform tasks such as object detection [16].

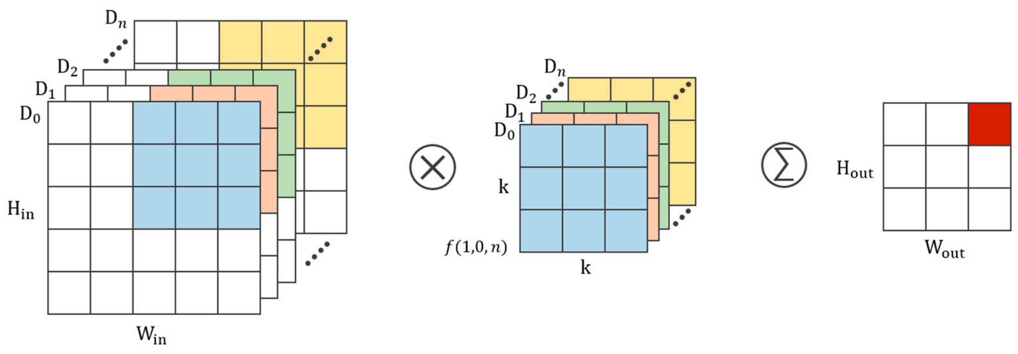

Figure 1. Operation of convolution.

2.2. RISC-V RV32I Architecture

RISC-V is an open-source instruction set architecture (ISA) designed according to the principles of the reduced instruction set computer (RISC). Its main features include simplicity, scalability, and modularity. Figure 2 below illustrates six representative instruction formats used in the 32-bit RISC-V ISA. Each instruction is structured according to specific formats, making them concise and intuitive, and enabling an efficient processor design.

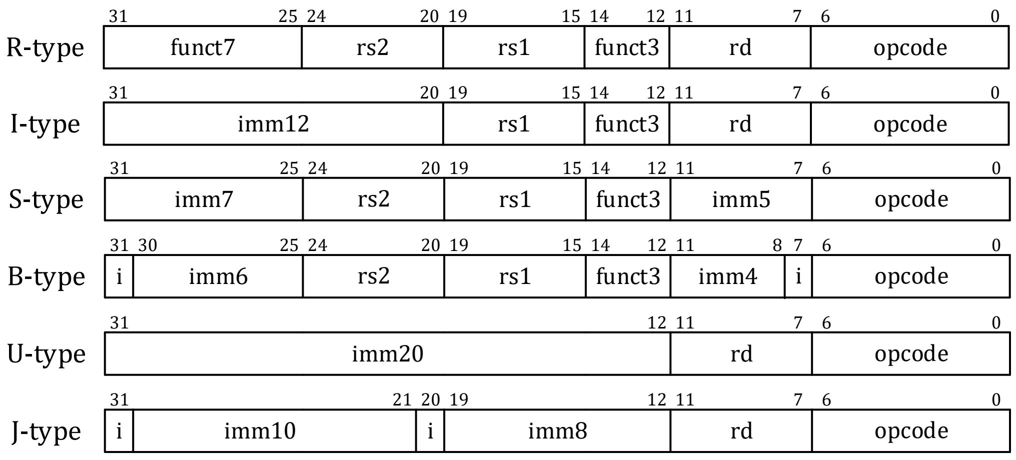
R-type: 31 funct7 25 | 24 rs2 20 | 19 rs1 15 | 14 funct3 12 | 11 rd 7 | 6 opcode 0
I-type: 31 imm12 20 | 19 rs1 15 | 14 funct3 12 | 11 rd 7 | 6 opcode 0
S-type: 31 imm7 25 | 24 rs2 20 | 19 rs1 15 | 14 funct3 12 | 11 imm5 7 | 6 opcode 0
B-type: 31 i | 30 imm6 25 | 24 rs2 20 | 19 rs1 15 | 14 funct3 12 | 11 imm4 8 | 7 i | 6 opcode 0
U-type: 31 imm20 12 | 11 rd 7 | 6 opcode 0
J-type: 31 i | imm10 21 | 20 i | 19 imm8 12 | 11 rd 7 | 6 opcode 0

Figure 2. RISC-V base instruction formats.

Table 1 lists key RISC-V instructions used prominently during compilation for convolution operations. Convolution involves multiplying a kernel with an image and accumulating the results into the existing output, commonly utilizing add and mul operations, as well as addi and slli instructions for memory address calculation [17]. Table 2 presents assembly code performing convolution operations using RISC-V instructions. The lw instruction loads kernel and image data from memory to the core, performs multiplication operations, and stores results in the memory area. Subsequently, it loads operation results from memory to the core using lw, performs addition operations, and stores results in the memory area. Repeating these steps by moving through memory areas computes the final convolution result. Figure 3 depicts the structure of a RISC-V core designed based on RV32I ISA. It comprises a five-stage pipelined structure: instruction fetch, instruction decoding, execute, memory, and write back. It includes control logic to manage the pipeline and generate control signals, an internal SRAM controller, and a load–store unit (LSU) to control peripheral addresses, a Harvard architecture with separate data and instruction memories, and an AXI4 (Advanced eXtensible Interface 4) system bus for bus systems. Such structures are widely used in low-area, low-power systems like the IoT, leveraging space-efficient designs and achieving a high performance through the Harvard architecture and pipelining techniques [18].

Table 1. Configuring the basic instruction of a convolution operation.

PIM Instruction		Instruction Format	Meaning
add	**rd, rs2, rs1**	R-type	R[rd] = M[rs2] + M[rs1]
mul	**rd, rs2, rs1**	R-type	R[rd] = M[rs2] $\times$ M[rs1]
slli	**rd, rs1, imm**	R-type	R[rd] = M[rs1] $\ll$ imm
addi	**rd, rs1, imm**	R-type	R[rd] = M[rs1] + imm

Table 2. Convolution operation into assembly instruction.

Assembly Instruction		Meaning
lw	**x14, −32(x8)**	Load data from address in memory R[x8]-32 to register x14
lw	**x14, −56(x8)**	Load data from address in memory R[x8]-56 to register x14
mul	**x15, x14, x15**	Multiply the data in registers x14, x15 and store the result in register x15
sw	**x15, −40(x9)**	Store data in register x15 to address in memory R[x9]-40
lw	**x14, −88(x9)**	Load data from address in memory R[x9]-88 to register x14
lw	**x15, −40(x9)**	Load data from address in memory R[x9]-40 to register x15
add	**x15, x14, x15**	Add the data in registers x14, x15 and store the result in register x15
sw	**x15, −40(x9)**	Store data in register x15 to address in memory R[x9]-40

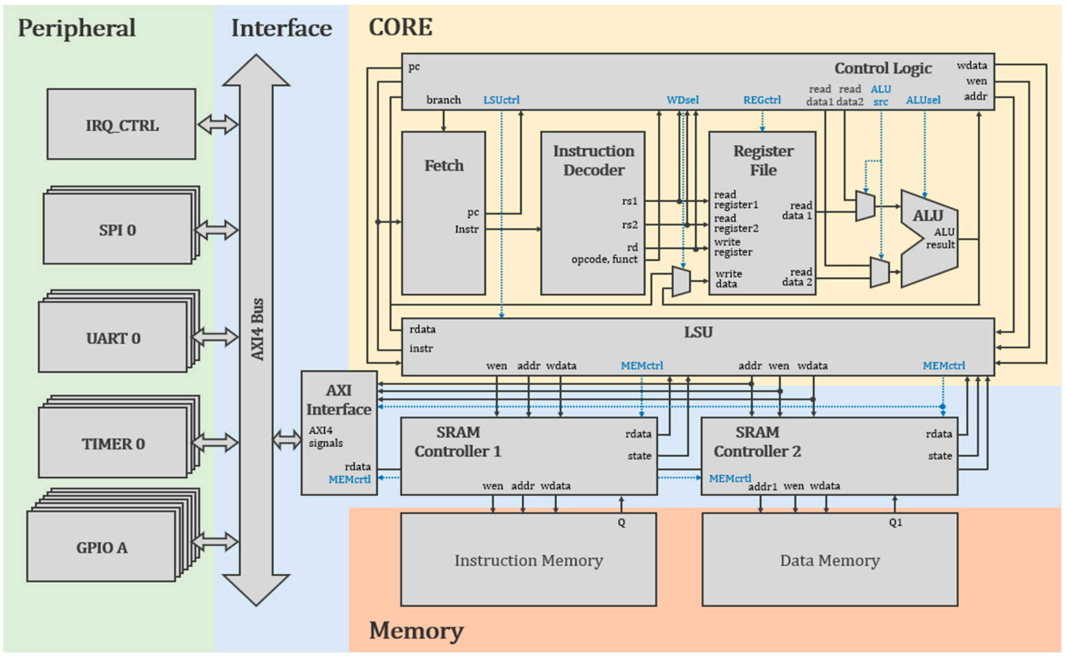

Figure 3. Block diagram of RISC-V RV32I architecture.

3. Proposed Design

In the existing von Neumann architecture, the core must fetch data from memory, perform computations, and then store the results back into memory. This process involves significant data movement, and systems targeting low-power and low-performance environments like RISC-V have limitations in bandwidth and speed between memory and processors due to various constraints such as power and area. These speed limitations during data movement lead to overall system degradation. In this paper, we propose to mitigate this issue by incorporating PIM capabilities into RISC-V systems to minimize the data movement between memory and processors.

We delineate methods for utilizing PIM systems in RISC-V from both software and hardware perspectives. First, for software-based PIM processing, we propose a method where PIM instructions are recognized as existing load instructions rather than introducing new instructions. This approach treats PIM instructions within the core as load instructions, enabling an efficient hardware design without significant alterations to the existing core system. Moreover, since they are recognized as load operations, compatibility with existing code structures is maintained without requiring additional decode logic, making it straightforward. Figure 4 illustrates the format of the proposed PIM instructions. To maintain compatibility with existing systems, we base the PIM-type on the existing I-type format of load instructions, dividing the original 12-bit immediate field into two 6-bit fields to enable the simultaneous access to two memory locations with PIM instructions. The positions of the opcode, funct, rd, rs1, and imm fields perfectly align with the existing I-type format, obviating the need for a separate decode logic. Table 3 shows the transformation of key instructions used in convolution operations (add, mul, slli, and addi) into PIM instructions, as add.p, mul.p, slli.p, and addi.p. The add.p and mul.pz instructions derived from add and mul instructions can operate on data from two memory addresses before loading, while slli.p and addi.p derived from slli and addi instructions can perform slli and addi operations on data from a single memory address before loading. Table 4 presents an example of converting convolution operations from Table 2 into PIM operations. In the conventional method, four instructions—two lw instructions to load the data from memory into registers, followed by either mul or add operations, and sw instructions to store results back to memory—are used. In contrast, the transformed instructions integrate two lw instructions and one operation instruction, recognized by the core as a single instruction (interpreted as load), and sw instructions. Consequently, using PIM instructions reduces the number of instructions the core processes from three to one, and allows parallel processing with PIM units, potentially achieving an up to 66% performance improvement in a core system with cycles per instruction (CPI) of 1. While significant performance gains over conventional instructions can be expected, representing a high number of register addresses and immediate values in a single instruction presents a trade-off between flexibility in address access and performance. Using a commercially available RISC-V compiler for compilation revealed the presence of immediate values exceeding the representation range (2^6), which cannot be processed by PIM instructions. Therefore, an essential PIM-aware compiler is required to effectively address such trade-offs. This proposal is not limited to convolution operation code but can also be applied to any code where two lw instructions and one computation instruction are consecutively arranged, or where one lw instruction and one computation instruction are consecutively arranged. However, due to the difficulty of demonstrating this transformation across all algorithms, this paper will explain its application specifically to CNNs as a representative example.

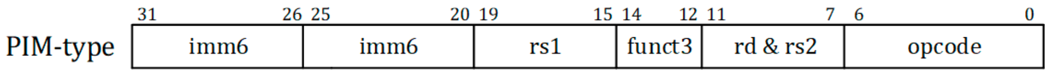

Figure 4. RISC-V base PIM instruction format.

Table 3. Configuring the basic PIM instruction of a convolution operation.

PIM Instruction		Instruction Format	Meaning
add.p	**rd, imm(rs1), imm(rs1)**	PIM-type	R[rd] = M[rs1+imm[25:20]] + M[rs1+imm[31:26]]
mul.p	**rd, imm(rs1), imm(rs1)**	PIM-type	R[rd] = M[rs1+imm[25:20]] $\times$ M[rs1+imm[31:26]]
slli.p	**rd, imm(rs1), imm**	PIM-type	R[rd] = M[rs1+imm[25:20]] $\ll$ imm[31:26]
addi.p	**rd, imm(rs1), imm**	PIM-type	R[rd] = M[rs1+imm[25:20]] + imm[31:26]

Table 4. Convolution operation into PIM assembly instruction.

Assembly Instruction		Meaning
mul.p	**x15, −32(x8), −56(x8)**	Multiply (in PU) the data in memory R[x8]-32, R[x8]-56 and load to register x15
sw	**x15, −40(x9)**	Store data in register x15 to address in memory R[x9]-40
add.p	**x15, −88(x9), −40(x9)**	Add (in PU) the data in memory R[x9]-88, R[x9]-40 and store to register x15
sw	**x15, −88(x9)**	Store data in register x15 to address in memory R[x9]-88

Secondly, to hardware-implement PIM instructions, we introduced PIM logic not inside the memory but in the SRAM controller, designing a control unit within the core to manage it. Figure 5 illustrates the RISC-V core structure for processing the proposed instructions. In the conventional von Neumann architecture system shown in Figure 3, processing units responsible for processing reside exclusively within the core. However, our proposed PIM structure incorporates a processing unit (PU) within the SRAM controller, enabling processing units to perform operations on read data from memory internally, transmitting only minimal data to the core. The PU added to the SRAM controller is designed to perform operations using read data from memory and does not fetch or decode instructions directly, necessitating the addition of a PIM control unit (PCU) within the core to decode PIM-related instructions and generate commands for executing operations in the PIM memory controller before the instruction is executed in the internal processing stage in the core. This unit generates PIM-transaction-related signals (PIMen, PIMsel, and PIMaddr) when instructions are fetched and decoded from the instruction memory to control the PIM logic. Additionally, one of its critical roles is to produce signals (pipeCtrl_pim2lw) to ensure that PIM instructions are processed similarly to lw instructions in the core's pipeline without additional pipeline stalls, smoothly integrating the process. Furthermore, it outputs lw-related flags to prevent the control logic from identifying PIM instructions as unknown instructions. This technique ensures that actual operations are performed not in the pipeline in the core but via instructions sent to PIM, simultaneously reading the data from memory. The control signals generated by PCU are utilized in the PU of the SRAM controller. The PU performs PIM operations using output Q1 and Q2 of the data memory. To facilitate this, a phase register is added to synchronize the SRAM read time and control signals, and a bypass mux is added to allow the memory data to be loaded directly without passing through the operation logic when PIM is not in use. This structure maintains compatibility with existing systems while enabling the efficient execution of PIM instructions through a streamlined design.

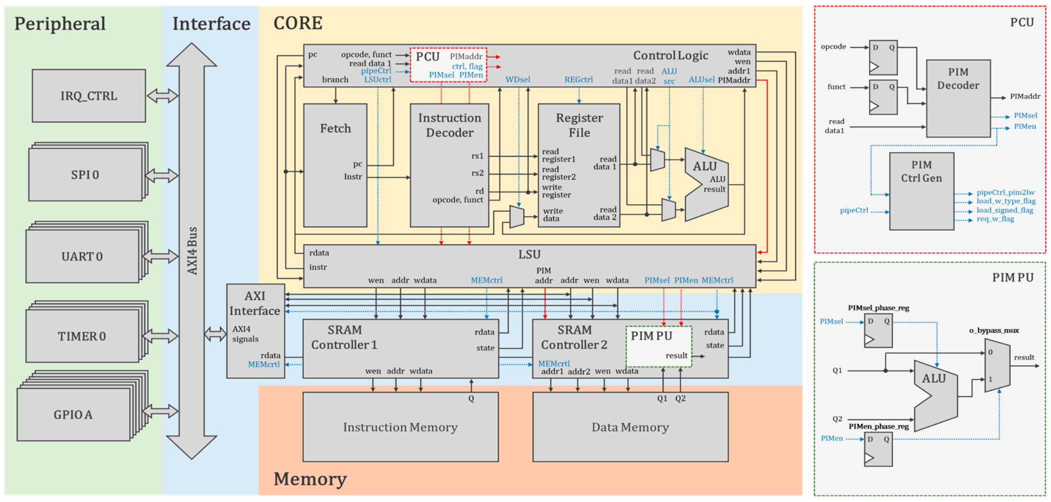

Figure 5. Block diagram of proposed architecture.

4. Experimental Results

To evaluate our proposed architecture, we assess the processing speed performance using convolution operations, which are the basic units used in deep-learning and large-scale data processing fields. Additionally, we compare the area when synthesized using the Synopsys Design Compiler for the CMOS 28 nm process at 100 MHz. We specifically compare the input data size of 224 × 224 × 3, typical for deep-learning algorithms targeting IoT devices like MobileNet, and three kernel sizes: 3 × 3, 5 × 5, and 7 × 7. The source code, originally written in C, is compiled using the RISC-V GNU Toolchain 12.1.0, with modifications to incorporate the PIM instructions proposed in this paper. This process is necessary due to the absence of compilers optimized for compiling and optimizing PIM instructions. Our experimental results demonstrate significant performance improvements in the proposed architecture featuring PIM instructions.

Firstly, in our proposed structure, the execution time of convolutions significantly decreases compared to traditional von Neumann-based architectures. Figure 6 compares the memory access rate for different kernel sizes between the proposed and existing structures. As a result, the memory access rate decreased by 24% when performing the convolution operation compared to the original. Figure 7 compares operation speeds for different kernel sizes between the proposed and existing structures. For each kernel size, the proposed structure showed a reduction in processing time of 31.4%, 32.7%, and 34.4% compared to the existing structure. This latency reduction is due to PIM instructions performing calculations directly in memory, reducing the number of instructions executed by the processor. While, theoretically, this could lead to a 66% performance improvement (as three instructions are consolidated into one), the actual compilation revealed limitations due to branch instructions and non-convertible codes, resulting in restrained performance gains. However, these results are derived from experiments without full compiler optimization, suggesting that more extensive use of PIM instructions could yield higher performance improvements.

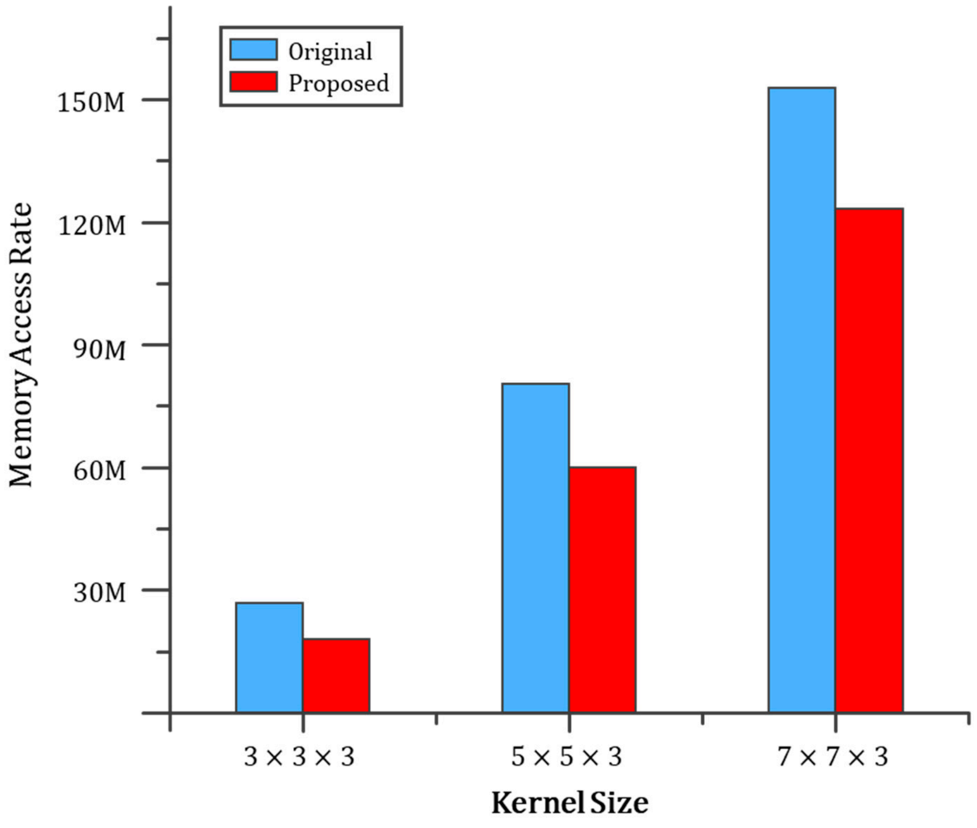

Figure 6. Comparison of memory access rate.

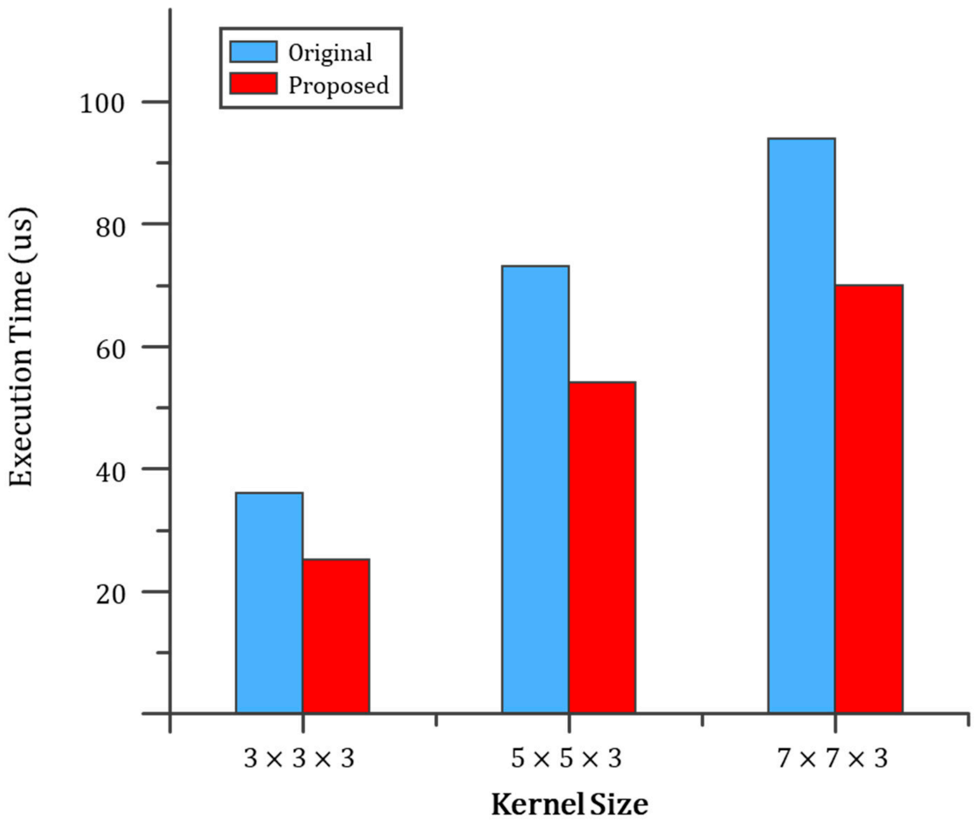

Figure 7. Comparison of execution time according to kernel size.

Secondly, we compare the synthesis results of the proposed architecture with the existing structure in terms of area. The proposed structure reduces data movement and the number of instructions processed in the core, enhancing the processing speed. However, implementing PIM systems requires additional logic to control PIM and perform operations in memory. This trade-off evaluates the high processing speed advantage against these requirements. Figure 8 compares the synthesis results to equivalent gate counts (normalized on a two-input NAND gate) between the proposed and existing structures. We evaluated areas divided into the core area with an added PCU, the interface (IF) area with an added PU, Memory (Mem), and Peripheral (Peri). The peripheral area remained the same; the proposed structure has increased in gate count by 285 for PCU and 5389 for PIM PU compared to traditional structures, resulting in a 1.27% increase in overall core and interface areas, leading to negligible area growth for significant performance improvements. However, there was a 31% increase in the memory area due to the additional usage in our system's dual-port RAM. If the proposed structure omits the dual-port RAM, the memory area implementation could match that of the existing structure, albeit with an additional cycle for loading memory data. This presents a choice for users between area and performance priorities. Through our experiments, we confirm that the proposed structure achieves an over 30% processing performance improvement with a modest 1.27% increase in the logic area. Figure 9 compares the energy consumption between the proposed and existing structures. Through our experiments, the energy consumption decreased by about 18% compared to the original.

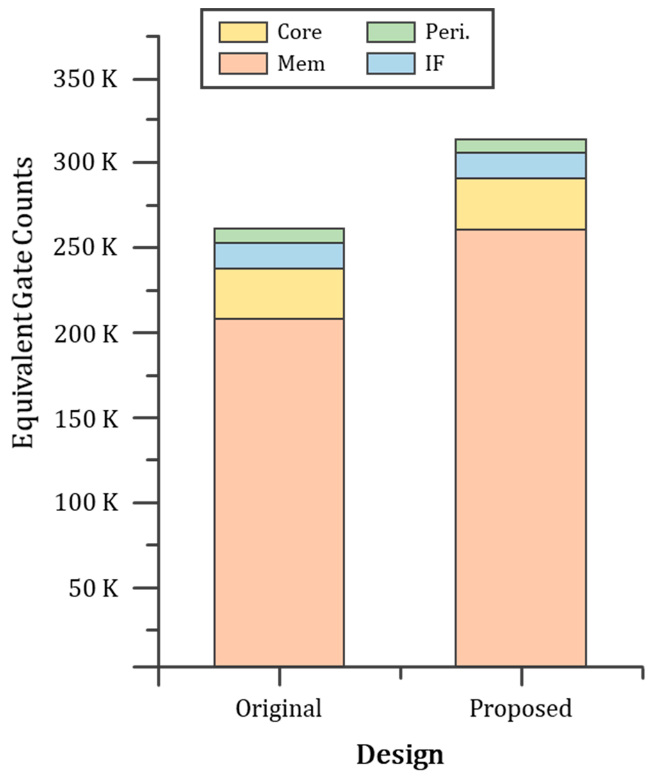

Figure 8. Comparison of equivalent gate count by design.

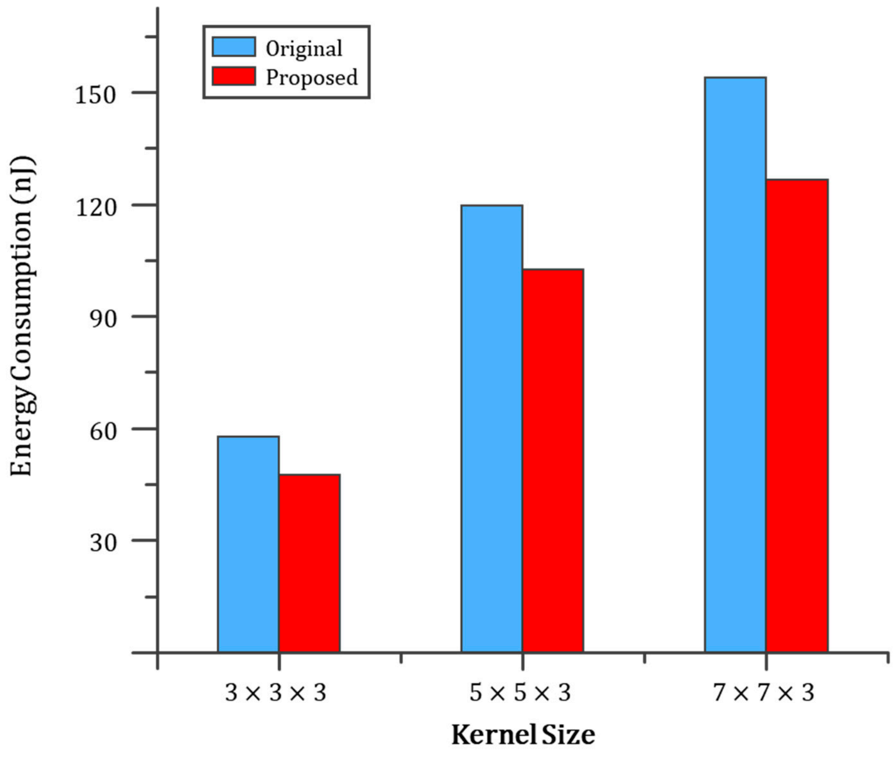

Figure 9. Comparison of energy consumption.

5. Conclusions

This paper proposes a PIM architecture as an alternative to traditional von Neumann architectures for efficient large-scale data processing in resource-constrained environments. The proposed system, based on RISC-V, introduces an innovative approach to efficiently handle large-scale data processing tasks in domains with limited computing performance. To evaluate this system, performance assessments are conducted on operations commonly used in large-scale data processing. In convolution experiments, we compare the processing speed of the proposed PIM system against a standard RISC-V processor. The results demonstrate how the PIM system alleviates bottlenecks associated with data movement. By performing computations directly within memory, the proposed system enhances the processing speed and reduces the latency associated with data movement. This insight suggests that leveraging PIM can significantly improve efficiency in scenarios where the data movement is a critical bottleneck. The findings of this paper provide important insights that open avenues for the development and application of PIM systems in future IoT and embedded environments. Effectively harnessing PIM will require support from the application software perspective. Subsequent research should focus on optimizing compilers capable of compiling new PIM instructions and developing optimized applications that utilize these instructions

Author Contributions: Conceptualization, J.L.; Software, J.L. and J.S.; Project administration, H.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

Acknowledgments: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2022R1A5A8026986). This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2022-0-01170).

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

References

1. Liang, D.S. Smart and Fast Data Processing for Deep Learning in Internet of Things: Less is more. *IEEE Internet Things J.* **2019**, *6*, 5981–5989. [CrossRef]
2. Zhuoying, Z.; Ziling, T.; Pinghui, M.; Xiaonan, W.; Dan, Z.; Xin, Z.; Ming, T.; Jie, L. A Heterogeneous Parallel Non-von Neumann Architecture System for Accurate and Efficient Machine Learning Molecular Dynamics. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2023**, *70*, 2439–2449.
3. Azriel, L.; Mendelson, A.; Weiser, U. Peripheral memory: A technique for fighting memory bandwidth bottleneck. *IEEE Comput. Archit. Lett.* **2015**, *14*, 54–57. [CrossRef]
4. Souvik, K.; Priyanka, G.; Jeffry, L.; Hemanth, C.; BVVSN, R. Memristors Enabled Computing Correlation Parameter In-Memory System: A Potential Alternative to Von Neumann Architecture. *IEEE Trans. Very Large Scale Intergration (VLSI) Syst.* **2022**, *30*, 755–768.
5. Cristobal, N.; Roberto, C.; Ricardo, B.; Javier, A.; Raimundo, V. GPU Tensor Cores for Fast Arithmetic Reductions. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *32*, 72–84.
6. Lee, J.; Kim, J.; Kim, K.; Ku, Y.; Kim, D.; Jeong, C.; Yun, T.; Kim, H.; Cho, H.; Oh, S.; et al. High bandwidth memory(HBM) with TSV technique. In Proceedings of the 2016 13th International SoC Design Conference (ISOCC), Jeju, Republic of Korea, 29 December 2016.
7. Park, I.; Singhal, N.; Lee, M.; Cho, S.; Kim, C. Design and Performance Evaluation of Image Processing Algorithms on GPUs. *IEEE Trans. Parallel Distrib. Syst.* **2011**, *22*, 91–104. [CrossRef]
8. Kim, D.; Yu, C.; Xie, S.; Chen, Y.; Kim, J.; Kim, B.; Kulkarni, J.; Kim, T. An Overview of Processing-in-Memory Circuits for Artificial Intelligence and Machine Learning. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2022**, *12*, 338–353. [CrossRef]
9. Lee, S.; Kang, S.; 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. In Proceedings of the 2021 ACM/IEEE 48th annual International Symposium on Computer Architecture, Valencia, Spain, 4 August 2021.
10. Lee, W.J.; Kim, C.H.; Paik, Y.; Kim, S.W. PISA-DMA: Processing-in-Memory Instruction Set Architecture Using DMA. *IEEE Access* **2023**, *11*, 8622–8632. [CrossRef]
11. Heo, J.; Kim, J.; Han, W.; Kim, J.; Kim, J. SP-PIM: A Super-Pipelined Processing-In-Memory Accelerator with Local Error Prediction for Area/Energy-Efficient On-Device Learning. *IEEE J. Solid-State Circuits* **2024**, *59*, 2671–2683. [CrossRef]
12. Elshimy, M.; Iskandar, V.; Goehringer, D.; Mohamed, A. A Near-Memory Dynamically Programmable Many-Core Overlay. In Proceedings of the 2023 IEEE 16th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, Singapore, 18–21 December 2023.

13. Dinelli, G.; Meoni, G.; Rapuano, E.; Fanucci, L. Advantages and Limitations of Fully on-Chip CNN FPGA-Based Hardware Accelerator. In Proceedings of the 2020 IEEE International Symposium on Circuits and Systems, Seville, Spain, 12–14 October 2020.
14. Heo, J.; Kim, J.; Lim, S.; Han, W.; Kim, J. T-PIM: An Energy-Efficient Processing-in-Memory Accelerator for End-to-End On-Device Training. *IEEE J. Solid-State Circuits* **2023**, *58*, 600–613. [CrossRef]
15. Krizhevsky, A.; Sutskever, I.; Hinton, G. Imagenet classification with deep convolutional neural networks. *Commun. ACM* **2017**, *60*, 84–90. [CrossRef]
16. Li, Z.; Liu, F.; Yang, W.; Peng, S.; Zhou, J. A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. *IEEE Trans. Neural Netw. Learn. Syst.* **2022**, *33*, 6999–7019. [CrossRef] [PubMed]
17. Wang, S.; Wang, X.; Xu, Z.; Chen, B.; Feng, C.; Wang, Q.; Ye, T. Optimizing CNN Computation Using RISC-V Custom Instruction Sets for Edge Platforms. *IEEE Trans. Comput.* **2024**, *73*, 1371–1384. [CrossRef]
18. Shin, D.; Yoo, H. The Heterogeneous Deep Neural Network Processor With a Non-von Neumann Architecture. *Proc. IEEE* **2020**, *108*, 1245–1260. [CrossRef]