ARES: A Mapping Framework of DNNs towards Diverse PIMs with General Abstractions

Xiuping Cui[1], Size Zheng[1], Tianyu Jia[1], Le Ye[1], and Yun Liang[1,2,*]

[1]Peking University, Beijing, China
[2]Beijing Advanced Innovation Center for Integrated Circuits
{cuixiuping, zhengsz, tianyuj, yele, ericlyun}@pku.edu.cn

2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD) | 979-8-3503-2225-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAD57390.2023.10323777

***Abstract*—Numerous architectures based on processing-in-memory (PIM) have recently emerged, exhibiting diversity in memory types, compute functions, memory mapping constraints, etc. To effectively utilize PIM hardware for deploying deep neural networks (DNNs), programmers face the challenge of mapping computations and data across multiple memory arrays, scheduling computation and data transfers, while adhering to various hardware constraints. Existing mapping approaches, however, are tailored to specific architectures and lack a general formulation for mapping optimization, limiting their applicability and performance.**

In this paper, we present ARES, a comprehensive mapping framework designed for diverse PIM architectures. The core of the framework is hardware abstractions for PIMs, which is inspired by the fact that DNNs on PIM hardware can be represented by a tensorized compute function and data layout constraints in the memory array. This abstraction forms the basis for constructing a mapping space that encompasses both compute and memory constraints. Through exploration of this mapping space, we derive efficient mapping strategies tailored to different PIM hardware configurations. Experimental evaluation conducted on four distinct hardware architectures demonstrates that compared to state-of-the-art mapping methods, ARES yields up to a 70% speed improvement for single operator mapping and a 50% speedup for overall network mapping.

I. INTRODUCTION

Recently, in-memory computing has emerged as a promising solution to address the memory wall problem. The placement of compute logic next to the memory array or its implementation inside the memory array can significantly reduce the demand for data transfer. Meanwhile, due to the rapid increase in computation demand, various Processing-In-Memory (PIM) architectures have been proposed to accelerate DNNs and reduce power consumption. For example, ReRAM-based PIM architectures, such as PUMA[1] and ISAAC[2] , as well as SRAM-based architectures such as Neural Cache[3] and Conv-RAM[4], employ the memory array itself as a computational array to perform the computation. By storing the weight and activation data of DNNs in the memory array, the latency and power consumption caused by off-chip data access can be reduced.

Based on the types of compute functions supported by PIM and the constraints of their data mapping, PIM hardware can be categorized into three types: Logic-based Processing Using Memory (PUM)[3], [5], Matrix-vector-multiplication-based (MVM-based) PUM[2], [1], [6], and Processing Near Memory (PNM)[7], [8]. Logic-based PUM treats the memory array as bulk bit-wise logic operation units, requiring strict column alignment for the operands. MVM-based PUM treats the array as a module for matrix-vector multiplication, where the rows and columns of the matrix correspond to the columns and rows in memory, respectively. PNM's compute functions are carried out by additional computation units such as reduction trees and SIMD processing units (PUs). The memory mapping requirements for these functions are more relaxed, without the need for strict row-column alignment in memory.

To map a deep neural network (DNN) onto a PIM accelerator, it is necessary to represent the computations of DNN operators with hardware compute functions and determine data layout within the memory array. Mapping is highly challenging for two main reasons. Firstly, there are numerous mapping approaches available. For example, when mapping a 2D convolution operator with 7 loops to a ReRAM array to perform matrix-vector multiplication using only inner product mode, there are 56 different mapping schemes. Secondly, computation and data mapping are tightly coupled in PIM hardware. Although two arrays can use the same compute function, their mapping rules can be different. As a result, both the shape and layout of the data in the mapping are different. For instance, we can map matrix-vector multiplication (MVM) to both MVM-based PIMs and logic-based PIMs. When mapping MVM to MVM-based hardware, the rows of the matrix are mapped to the columns of the crossbar, and the columns of the matrix are mapped to the rows of the crossbar. On the other hand, when mapping it to logic-based PUMs, the matrix data can be unfolded and placed in a single row of memory, intermediate results can be obtained through bulk multiplication operations, and then the intermediate results can be transferred and reduced between columns.

Prior mapping solutions[1], [3] adopt template-based mapping and are limited in terms of both generality and efficiency. First, they can only target a specific type of hardware and a specific class of DNN operators, requiring researchers to design mapping approaches for each hardware and operator separately. For instance, the work [3] presents a mapping approach for customizing 2D convolution operators on an SRAM-based PIM accelerator Neural Cache. However, this

* Corresponding author

specific mapping approach is not universally applicable across other PIM architectures. Second, template-based mapping is not optimal as it is designed for a few fixed mappings only. For example, PUMA[1] maps weights to crossbar by mapping the input-channel and output-channel dimensions to the row and column dimensions of the crossbar, respectively. They don't consider mapping input feature maps to the crossbar or mapping the same kernels to multiple crossbars to accelerate computation.

We find that despite the various designs of PIMs, their core functionalities can be represented by a tensor-based compute function and data mapping constraints. For instance, PUMA[1] and ISAAC[2] are PIM hardware implementations based on ReRAM, where the compute function is matrix-vector multiplication (MVM), and the mapping of matrix elements involves mapping rows (columns) of the matrix to columns (rows) within the array. When performing operator mapping, the key point is to represent the operator through hardware compute functions, while concurrently adhering to the constraints imposed by data mapping. Therefore, we propose a hardware abstraction around the core of PIM and develop a mapping framework for diverse PIM platforms based on this abstraction.

In this paper, we develop ARES, a general mapping framework for diverse PIM platforms. The core of this framework is a general hardware abstraction for PIM platforms. The hardware abstraction consists of two aspects: compute abstraction and memory abstraction, corresponding to the hardware's compute functions and data mapping constraints, respectively. Compute abstraction includes two parts: arithmetic types and the form of tensor computation. Memory abstraction can be represented using a mapping matrix and an offset vector. We perform mapping by jointly scheduling computation and memory mapping. Through an analysis of compute abstraction, we derive valid mapping strategies involving the correspondence between tensors within operators and hardware tensor operands, as well as the mapping between operator loop dimensions and hardware compute dimensions. Subsequently, constrained by the abstractions of memory, we further ascertain constraints on loop sizes mapped onto the hardware. By combining compute and memory abstractions, we establish an exhaustive set of feasible mapping schemes.

Building upon this foundation, we orchestrate the scheduling of operator computations on the Processing-in-Memory (PIM) architecture, encompassing considerations such as loop orders and the utilization of memory arrays, thereby generating the comprehensive mapping space. Upon exploring the feasible mapping space using dataflow models[9], [10], [11], [12], [13], we can find the optimal mapping strategies. Leveraging the aforementioned methodology, we actualize an automated mapping framework. By soliciting operator descriptions within the network and hardware abstractions from the user, our framework builds up the mapping space, generates feasible mapping alternatives, and systematically explores these alternatives within the constructed mapping space to yield highly efficient mappings.

The contributions of this paper are summarized as follows:

- We propose a general abstraction for PIM devices that includes compute abstraction and memory abstraction. This hardware abstraction can model the scheduling space of various PIM devices.
- We build a systematical mapping space for PIM devices, which enables the mapping of different tensor operators onto diverse PIM devices.
- We build an automatic mapping flow. Using this flow, we design a mapping framework that efficiently maps neural networks to diverse PIM platforms.

We conduct experiments on four different hardware platforms adapted from real-world PIM accelerators, including SIMDRAM[14], Neural Cache[3], PUMA[1] and PIM-HBM[7]. Compared to the fixed template-based mapping approaches proposed in [15], [1], [7], our results achieve up to 70% speedup for single operators and 50% speedup for overall network mapping.

II. BACKGROUND

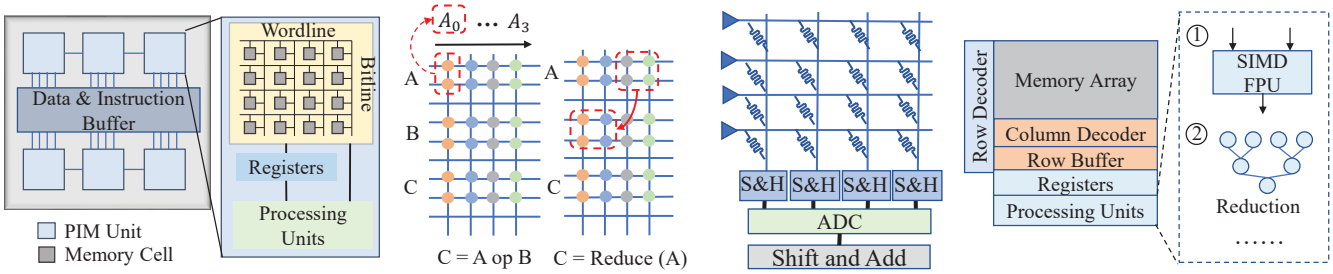
PIM devices can be represented uniformly as illustrated in Figure 1a. A PIM unit is defined as a complete computational logic that can directly access an array of memory, which is the fundamental component of PIM devices. The PIM unit integrates the functionality of both memory and computation, and is composed of memory arrays, registers or buffers, processing logic and control units. The implementation of these modules may vary in different hardware. PIM units are connected via a bus or on-chip network, and can receive data and instructions from buffers. Together, these components constitute the entire PIM device. PIM devices can interact with host devices, such as CPUs and GPUs, and access data from off-chip host memory, such as DRAM and NVM.

The PIM architecture can be divided into three categories, including logic-based PUM, MVM-based PUM, and PNM.

A. Logic-based PUM

Logic-based PUM performs logic operations by exploiting the circuit properties of memory arrays[5], [16], [17], [14], [3], [18], [19], [20], [21]. Different memory types or initialization methods can result in various logic operation types, such as NOR and NAND. The basic unit of operation is a row, and for binary operations like AND and OR, or unary operations like NOT, multiple rows that hold operands are simultaneously activated when computing. The operation is element-wise, and data on the same bitline are operands. As the basic operation is a one-bit logic operation, bit-serial compute mode is used to implement advanced operations such as multi-bit addition and multiplication. This approach requires all bits of the operands to be placed in the same column of the memory array (rather than the same row), as depicted in Figure 1b left, and computation is performed from low bit to high bit.

In SRAM- and ReRAM-based circuits[3], [22], data transfer across bitlines is typically designed to support reduction operations on data from different bitlines, as illustrated in Figure 1b. We assume that data transfer across bitlines is also



(a) PIM hardware overview (b) Logic-based PUM (c) MVM-based PUM (d) processing-near-memory (PNM)

Fig. 1: Processing-in-Memory Architecture

feasible in DRAM-based implementations. This assumption can be implemented using the design proposed in Drisa[17] or by adding auxiliary circuits besides the sense amplifiers.

B. MVM-based PUM

MVM-based PUM utilizes a memory array to perform MVM operations [1], [6], [23], [4], [2]. In this approach, the matrix elements are stored in the memory array prior to the operation, and the vector is input along the wordline. Multiplication operations are then performed between the input data and the data stored in memory cells. Addition operations are carried out on the bitline based on Kirchhoff's law or additional adder trees to obtain the final MVM result. Each cell in the memory array can store 1-bit or multi-bit data. If the weight data exceeds the capacity of one cell, the data is placed in adjacent cells on the same wordline, and the final results are obtained by combining adjacent results through shift-and-add units. If the input data width exceeds the bit width limit of each wordline or precision of digital-analog converters, the input data is provided in multiple cycles in a bit-serial manner, and the data from multiple cycles are accumulated using shift-and-add units to obtain the final result.

C. Processing-near-Memory

Near-memory computing [24], [25], [26], [27], [28], [29], [30], [31], [32], [8], [7], [33] is a computing paradigm that enables flexible compute logic implementations and memory mapping rules compared to PUM. Its computation is achieved by embedding additional computation circuits within the memory hierarchy, as illustrated in Figure 1d.

The embedded circuitry can realize various operations, such as SIMD vectorization[7], reduction trees, dot operations[30], or a function-complete processing core[8], such as a RISC Core. There are three main differences between near-memory computing and traditional computing systems. First, because near-memory computing lacks a memory controller, data placement in memory needs to be more precisely controlled. Second, the computational logic can be embedded in different locations in memory, such as the rank dimension of 2D-DIMMs, the bank dimension, the logic die of 3D stacked memory, etc. Different embedded compute circuits require different data placement methods. Third, the data path in near-memory computing is more complex, for instance, PIM-HBM's computation unit can obtain operands from both memory and registers, and the data in registers can come from the adjacent memory banks or be directly written into the PIM units from the host processor. Therefore, careful consideration is required to decide which data should be stored in the register and which data should not be placed in the PIM unit to ensure efficient operation.

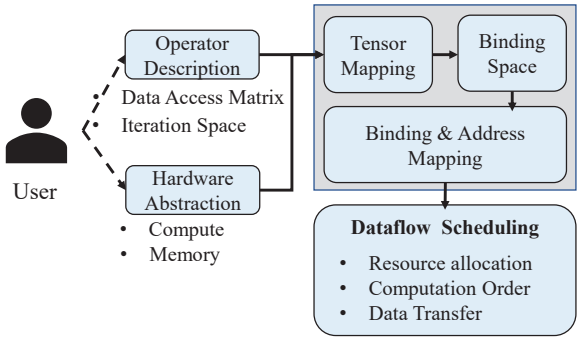


Fig. 2: ARES working flow

III. ARES OVERVIEW

The workflow of ARES is shown in Figure 2. The user's input consists of two parts: high-level operator descriptions and hardware descriptions. The operator description mainly includes the iteration space of the operator and the access matrixes of tensors, which specify the relationship between the computations performed by an operator and the corresponding data elements accessed during those computations. The hardware description is written using our proposed hardware abstractions: compute abstraction and memory abstraction. Compute abstraction represents the compute function of the PIM hardware, while memory abstraction represents the mapping requirements of the data in compute abstractions.

The hardware abstraction is explained in detail in Section IV. ARES generates mappings mainly through two steps. The first step is compute binding, which involves representing the software computations using hardware compute functions. In this step, we first map software tensors to hardware tensors. Then, based on the analysis of compute abstraction and data

mapping matrix in memory abstraction, we generate feasible loop binding options. Finally, combining with memory abstraction and the size of the memory array, we generate constraints on the size of binding loops and data mapping functions. The second step involves scheduling analysis for resource allocation, execution order of compute functions, and data transfers between different levels of memory. These steps are described in detail in Section V. Through the above steps, we can obtain the efficient mappings for various PIM backends.

IV. Hardware Abstraction

This section explains the uniform hardware abstractions, which consist of compute abstraction and memory abstraction. Compute abstraction models operators that the hardware supports, including both arithmetic types and tensorized form. Memory abstraction models constraints of data layout, which must be met to perform computation on hardware.

A. Compute Abstraction

Definition 1: **Compute Abstraction.** Compute abstraction is an equation that delineates the hardware compute function, establishing a correspondence between the output tensor and input tensors. Given the arithmetic type denoted as F, the compute dimension space denoted as $\vec{D}$, and memory access indices $\vec{i}$ and $\{\vec{j_0}, \vec{j_1}, ...\vec{j_{k-1}}\}$ for the output operand and input operands, respectively, the formulation is as follows:

$$Dst[\vec{i}] = F(Src_0[\vec{j_0}], ..., Src_{k-1}[\vec{j_{k-1}}]) \tag{1}$$

In this formulation, the output tensor element at index $\vec{i}$ is computed based on the arithmetic operation F applied to elements $Src_0[\vec{j_0}], ..., Src_{k-1}[\vec{j_{k-1}}]$ from input tensors.

The arithmetic type F describes high-level operations used in network operators such as multiply-add and activation operations, which are not low-level operations natively supported by the hardware. If the required high-level operation is not directly supported by the hardware primitives, then additional steps are needed to generate the arithmetic type for the operation. The vector $\vec{D}$ is used to determine the dimensions of computation. Unlike traditional hardware primitives, the range of $\vec{D}$ cannot be directly determined by an affine transformation operation. The range of $\vec{D}$ is related to the size of the memory array used for computation, the implementation method (bit-parallel or bit-serial), and other factors. Constraints on the range of $\vec{D}$ must be established after defining the memory abstraction.

B. Memory Abstraction

Definition 2: **Memory Abstraction.** Memory abstraction are functions that delineate the bit-level mapping constraints of tensor operands within a compute abstraction. The input to each function consists of the position of a bit unit within the hardware computational space, encompassing two distinct components: the index $\vec{j}$ of the hardware compute dimension and the position b of the bit within an element of the tensor operand. The output is a position vector, signifying the location of the specified bit of the tensor element within the memory array. The function can be modeled as a linear function, which can be represented in the following form:

$$\begin{aligned} \vec{addr} = f(\tilde{j}) = & A \cdot \tilde{j}^T + \vec{offset}, \text{ for all } b : 0 \le b < B \\ & \tilde{j} = [\vec{j}, b] \end{aligned} \tag{2}$$

where $\vec{addr}$ is the memory address constraint of each bit of an element in tensor T. $\tilde{j}$ is the augmented index of the tensor, which is composed of original index $\vec{j}$ in the tensor and bit index b in an element. Matrix A is the memory mapping matrix that maps each bit to a memory location. $\vec{Offset}$ models the free variables in the mapping procedure. For one compute abstraction instance, the $\vec{offset}$ is a constant vector, which does not need to be the same across multiple compute instances.

C. Discussion

Since the main characteristics of PIM hardware can be represented by a tensor-based compute function and corresponding address mapping constraints, our proposed hardware abstraction can capture the diversity of PIM hardware. We provide two examples to illustrate this.

1) Case I: MVM-based PUM: The compute and memory abstraction of MVM-based PUM can be represented as follows:

$$\begin{aligned} Dst[m] &= MAC(Src_1[m,k], Src_2[k]) \\ addr_{Src_1} &= \begin{pmatrix} 0 & 1 & 0 \\ B & 0 & 1 \end{pmatrix} * (m,k,b)^T + (o_1, o_2)^T \end{aligned}$$

The compute abstraction is a matrix-vector multiplication operation. In the memory abstraction, the reduction dimension k is mapped to the row dimension of the memory array, while the parallel dimension is mapped to the column dimension of the memory array. Each element in the Src_1 matrix occupies adjacent B memory cells.

2) Case II: Logic-based PUM: The compute and memory abstraction of logic-based PUM can be represented as follows:

$$\begin{aligned} Dst[m] &= MAC(Src_1[m,k_1,k_2], Src_2[m,k_1,k_2]) \\ addr_T &= \begin{pmatrix} 0 & x & 0 & 1 \\ 1 & 0 & M & 0 \end{pmatrix} * (m,k_1,k_2,b)^T + (o_1, 0)^T \end{aligned}$$

The compute abstraction includes one parallel dimension $\{m\}$ and two reduction dimensions $\{k_1, k_2\}$. The parallel dimension describes the basic SIMD operations. Logic-based PUM involves bitwise reduction operations, and the k_1 and k_2 reduction dimensions are used to model these operations. k_1 represents the temporal dimension of reduction operations, which are performed sequentially for k_1 iterations. k_2 represents the spatial dimension of reduction operations. After k_1 SIMD operations, we need to reduce k_2 groups of elements to 1 group. This requires $log_2(k_2)$ reduction operations.

During the mapping process, the elements corresponding to m and k_2 are placed in the column dimension of the memory array, while the elements corresponding to k_1 are placed in the row dimension of the memory array. Due to the bit-serial

execution mode, the bits corresponding to the same element also need to be placed in a column. It is worth noting that there is a symbolic variable x in the mapping matrix. This variable indicates that the data corresponding to the k_1 dimension should be placed in the row dimension, but it does not require continuous placement.

V. Mapping Flow

This section first defines the mapping problem for in-memory computing hardware. Then, it introduces how to construct a complete mapping space. Finally, we introduce our approach for exploring the mapping space.

A. Mapping Definition

Mapping is the transformation of DNN operator computations into invocations of hardware compute functions. This process can be divided into two main steps: binding, which entails determining the computation carried out by an individual compute function, and scheduling, which entails establishing the sequential order and spatial arrangement of multiple compute function invocations.

Given the set of operands $\{T_i\}$, encompassing both source and destination tensors, the loop iterations $\{L_i\}$, and the data access matrixes $\{A_i\}$ associated with a deep neural network (DNN) operator, the computation of the operator can be expressed as $(\{T_i\},\{L_i\},\{A_i\})$. Similarly, considering the operands $\{O_i\}$, compute dimensions $\{D_i\}$, data indices in the hardware compute abstraction $\{I_i\}$, and the mapping matrixes $\{M_i\}$ within the memory abstraction, the computation performed by an individual processing-in-memory (PIM) unit can be represented as $(\{O_i\},\{D_i\},\{I_i\},\{M_i\})$. The binding process can be further subdivided into three types of mappings: operand binding, dimension binding, and binding size determination. These mappings are elaborated as follows:

$$\begin{aligned} &\{T_i \rightarrow O_j\} \\ &\{L_i \rightarrow D_j\} \\ &\{L_i \rightarrow S_i\} \end{aligned} \tag{3}$$

In the given equations, S_i represents the size of the loop L_i binding to the hardware compute dimension O_j.

After mapping the software computations onto hardware compute functions, we further schedule these hardware primitives, entailing the determination of the execution position and timing for each individual hardware primitive. This can be formalized as follows:

$$\{L_i \rightarrow (p_i, time_i)\} \tag{4}$$

where p_i signifies the spatial location of the memory array, while $time_i$ represents the begin time of the associated hardware function.

B. Mapping Space

In the subsequent exposition on the mapping space, we employ the mapping of 2D convolution onto MVM-based PUM as an illustrative example as in Figure 3 to elucidate

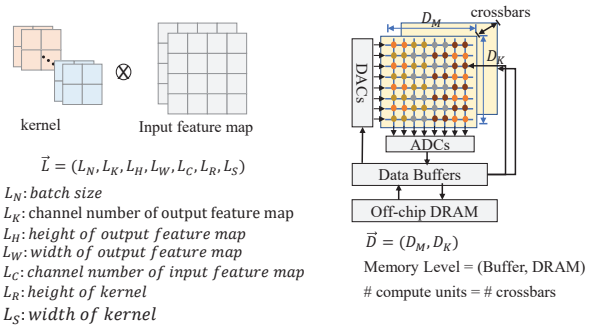


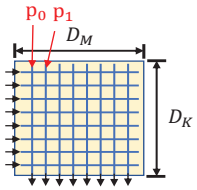
Fig. 3: Convolution 2D workload and architecture of MVM-based PIM hardware

our mapping process. Figure 3 depicts the 2D convolution operation alongside the target hardware. The 2D convolution operator encompasses seven nested loop iterations, representing distinct dimensions such as batch, channel of output feature map, output feature map's length and width, channel of input feature map, as well as the dimensions of the convolution kernel's length and width. As explained in Section IV-C1, the hardware compute function exhibits two key dimensions, namely D_M and D_K. The hardware's computation is performed by *#crossbar* memory arrays, while the memory hierarchy encompasses two levels, encompassing on-chip data buffers and off-chip main memory (DRAM).

The overall mapping space can be partitioned into two main components: binding and scheduling. Binding involves the mapping of network operators onto hardware compute functions, while scheduling entails the orchestration of these compute functions, determining execution order, spatial location, and the number of memory arrays engaged in the computation, among other factors.

1) Binding: The core part is to determine the latter two mapping in Equation 3. We first classify the hardware dimension types. We can divide them into parallel dimensions and reduction dimensions. Parallel dimensions calculate different output tensor elements, while reduction dimensions calculate the same output tensor element. In Figure 3, D_M is a parallel dimension, and D_K is a reduction dimension. A basic idea is to map the parallel (reduction) dimensions in the operator's iteration space to the parallel (reduction) dimensions in the hardware, that is, $\{L_N, L_K, L_H, L_W\} \rightarrow D_M, \{L_C, L_R, L_S\} \rightarrow D_K$.

The mapping approach above may not always ensure correctness. This is because the data reuse in the compute abstraction and the data reuse rules of software operators are different. Assume that tensor T_{i_1} in the operator is mapped to O_{j_1}, and dimension L_{i_2} is mapped to D_{j_2}, if T_{i_1} is not reused in dimension L_{i_2} while O_{j_1} is reused in dimension D_{j_2}, then the mapping would be incorrect. We illustrate this with the example in Figure 4. Figure 4 illustrates two different binding approaches. In the first binding approach, the weights are mapped onto the crossbar, and the L_H and L_C dimensions correspond to the hardware's D_M and D_K dimensions, respectively. With this binding approach, the data



Weight:
W_{L_k,L_C,L_R,L_S}
Input Feature Map:
$F_{L_N,L_C,L_H+L_R,L_W+L_S}$

Binding	Compute at p_0 and p_1
$\{Weight\} \rightarrow Crossbar$ $\{L_H\} \rightarrow D_M$ $\{L_C\} \rightarrow D_K$	$C_{p_0} = W_{0,0,0,0} * F_{0,0,0,0}$ $C_{p_1} = W_{0,0,0,0} * F_{0,0,1,0}$
$\{Weight\} \rightarrow Crossbar$ $\{L_K\} \rightarrow D_M$ $\{L_C\} \rightarrow D_K$	$C_{p_0} = W_{0,0,0,0} * F_{0,0,0,0}$ $C_{p_1} = W_{1,0,0,0} * F_{0,0,0,0}$

Fig. 4: Examples of conv2d operator to MVM-based PIM hardware

corresponding to the input feature map needs to be accessed along the wordlines, and memory cells on the same wordline need to access the same data of input feature maps. Taking the computations at positions p_0 and p_1 in the figure as examples, the computation at p_0 is $W_{0,0,0,0} * F_{0,0,0,0}$, while the computation at p_1 is $W_{0,0,0,0} * F_{0,0,1,0}$. Since they require different input feature map data, this mapping approach is incorrect. On the other hand, in the second mapping approach depicted, all the memory units on the same wordline utilize the same data elements of the input feature map. Therefore, this mapping approach is correct. To get the correct binding approaches, we need to determine the dimensions in which a tensor can be reused.

For a data T_i, it can be reused in a dimension L_j if and only if L_j does not participate in the data indexing computation. In other words, the sum of absolute values in column L_j of its access matrix A is equal to zero, which can be expressed formally as follows:

$$\sum_k (|A_{k,j}|) = 0 \tag{5}$$

By applying this, we can clearly determine all feasible loops $\{L_i\}$ that can be mapped to hardware dimension D_j.

We take the mapping of 2D convolution to MVM in Figure 3 as an example. When the weights of the convolution are mapped to crossbar, the feasible mapping strategy is

$$\{L_C, L_R, L_S\} \rightarrow D_K, \{L_K\} \rightarrow D_M$$

and when the input feature map of the convolution is mapped to crossbar, the feasible mapping strategy is

$$\{L_C, L_R, L_S\} \rightarrow D_K, \{L_N, L_H, L_W\} \rightarrow D_M$$

Based on the feasible mapping strategies, we need to select a specific mapping scheme and determine the size of the mapping iterations with respect to the constraints of the computing units and memory arrays. When selecting a mapping scheme, we assume that each loop can be mapped to at most one hardware dimension, and the mapping size of L_i is S_i. We take the MVM-based PIM devices for example here. Assuming the memory size limit is (X,Y), when we map input feature to the crossbar and each element in the input feature map occupies B memory cells, the constraints of the mapped iteration size is as follows:

$$S_C * S_R * S_S \leq X$$
$$S_N * S_H * S_W * B \leq Y$$

2) Scheduling: After compute binding, we determine the computation performed by a single compute function. In order to fully map the operator onto PIM hardware for computation, we need to schedule PIM units and all levels of data buffers, that is, dataflow scheduling. These scheduling operations correspond to loop transformations such as tiling, reordering, and parallelization. For each level of the hierarchical data buffers, we need to map a sub-space of iteration space and determine the execution order of this sub-space, which has a big impact on the size of data transfer. If there are multiple compute modules, we can also map some loops to spatial dimensions to fully utilize the computing resources.

C. Mapping Space Exploration

We employ a genetic algorithm for exploring the mapping space. We treat loop binding, sizes of loops mapped to spatial and temporal dimensions as genes. Each mapping can be represented as a genome. By applying transformations such as crossover and mutation to the genes, we can generate new potential genes based on the existing ones. The crossover operation happens between two parent genomes, interchanging the value of tile sizes. Mutation occurs within a genome by changing the tile size, loop orders and parallel dimensions. The fitness is defined as the execution time on the hardware simulator. By these genetic transformations, we can explore the mapping space efficiently to find the optimal mapping.

VI. EXPERIMENT

A. Experiment Setup

We choose three kinds of PIM hardware as our experimental platforms, logic-based computing devices, MVM-based devices and near memory computing devices. These platforms can cover most of the computing paradigms of PIM. For logic-based PIM devices, we select logic-{DRAM, SRAM} as

TABLE I: Logic-DRAM computing system parameters

Processor	14 cores/28 threads, 2.6GHz L{1-3} cache: 32KB, 256KB, 35MB
L3 cache	14 slices, 20 ways, 4 banks, 4 arrays Array size: 256 * 256
Memory controller	8KB row size, FR-FCFS scheduling
Main memory	DDR4-2400, 4 channels, 4 ranks, 16 banks Timing parameters: tRCD = 17, tCL = 17, tRP = 17,tRC = 56, tCCD_S=4, tCCD_L=6,tRRD_S=4, tRRD_L=6, tFAW=26, tBL=4,

TABLE II: In-ReRAM computing architecture parameters

Hierarchy Structure	1 node - 4 * 4 tiles - 8 cores - 2 crossbars
Crossbar	size: 128 * 128
Cores	register size:1KB connection: bus
Tile	shared memory: 64KB connection: mesh

TABLE III: Near-DRAM computing Parameters

Main memory	HBM2, 8GB, 32 pseudo channels 8 memory dies (4 with PUs)
NDP	32 adder, 32 multiplier, 300MHz registers: GRF x32 (each 32 * 8b)

our platform. We select an Ambit-like [5] architecture design as logic-DRAM platform, which is based on the existing DDR4 hardware, and the hardware parameters are available in industry datasheets [34], shown in Table I. We select Neural cache-like architecture design as logic-SRAM platform, which is implemented based on the last level cache of Intel Xeon E5-2697 v3. The hardware parameters are from the work Neural Cache[3]. For MVM-ReRAM devices, we implement a hardware like PUMA [1], and hardware parameters are from PUMA, shown in Table II. For near-DRAM computing paradigm, we implement a hardware platform similar to PIM-HBM [7], where the compute logic is implemented on bank-level on 4 memory dies of HBM2, and the hardware parameters are from [7], shown in Table III. We develop cycle-accurate simulators for the these hardware platforms.

For the three kinds of hardware, we select different baseline mapping strategies. For logic-PIM devices, we use the fixed-template mapping approachs in Neural Cache. They map kernel height (R) and width (S) to one bitline, and map input channel (C) dimension to multiple wordlines. And C bitlines are responsible for a few points in a single feature map of one output channel (K) dimension. For MVM-PIM, we choose an channel-wise binding approach [1]. Channel-wise binding maps input channels and output channels to row dimension and column dimension of crossbars, respectively, which is a kind of weight-stationary mapping. For near-memory computing devices, we take a fixed binding approach as our baseline. For 2d-convolution, we map the output channel dimension to the hardware primitive; for GEMM kernel ($C[M,N] = A[M,K] * B[K,N]$), we map M or N dimension to the hardware primitive.

We conduct experiments on a wide range of DNN workloads to verify the efficiency of mapping under different operator configurations. The networks we select include ResNet50 [35], ResNeXt-50 [36], MobileNet [37], Bert[38], GoogleNet[39], DenseNet[40]. In addition to evaluating the entire network, we also conduct tests on individual convolution-2d operators and gemm operators. The convolution-2d operators are selected from ResNet18, and their shapes are presented in Table IV. The shapes of gemm operators are [1024,1024,1024] (denoted as $G1$) and [2048,2048,2048] (denoted as $G2$) for $[M,K,N]$.

TABLE IV: Configurations of the single operators

Layer	N	K	H	W	C	R	S	Stride
C0	4	64	112	112	3	7	7	2
C1	4	64	56	56	64	3	3	1
C2	4	64	56	56	64	1	1	1
C3	4	128	28	28	64	3	3	2
C4	4	128	28	28	128	1	1	1
C5	4	128	28	28	128	3	3	1
C6	4	256	14	14	128	3	3	2
C7	4	256	14	14	256	1	1	1
C8	4	256	14	14	256	3	3	1
C9	4	512	7	7	256	3	3	2
C10	4	512	7	7	512	3	3	1

For all the networks and operators, the precision is 8-bit for both weights and input data.

B. Results of Logic-PIMs

1) Single Operator Results: Figure 5 shows the performance results of our method and baselines. The performance improvements are 20.13% and 27.98% on average for logic-SRAM and logic-DRAM, respectively. For different layers, the performance improvements comes from different aspects. For operators that have larger sizes of spatial dimensions, hardware's compute resources are fully utilized, and how to schedule data in and out to reduce data transfer traffic dominates the performance; for operators that have smaller sizes of spatial dimensions, how to parallelize the computation to make better use of computation resource has a more significant impact on the final performance.

2) Full Network Results: Figure 6 shows the performance of the full networks. For an entire network, there is an overall performance improvement of up to 21% and 26%, for logic-SRAM and logic-DRAM, respectively. The performance improvements varies widely across networks. For example, for logic-SRAM devices, the resnet shows only 11% improvement; while for the vgg net there are 21% improvement.

C. Results of MVM-PIM

1) Single Operator Results: Figure 5 is the performance results of our method and baselines. The performance improvements are 69.2% for MVM-ReRAM platform. The performance improvements comes from two aspects. The first is that in our schedules, we can copy weight kernel to multiple crossbars, which greatly increases the parallelism; the second is that by scheduling input and output data flow, we reduce the data traffic.

2) Full Network Results: For the whole network schedule on MVM-ReRAM platform, there's one more schedule dimension compared to single operator scheduling. The new dimension is that we need to allocate resources properly among layers of the whole network. If the layer sequential execution is used directly, we can minimize the execution time of individual operators, but we need to keep loading weights when computing different operators. The time of crossbar programming is far more than one MVM operation. Therefore, the execution mode of layer sequential is not

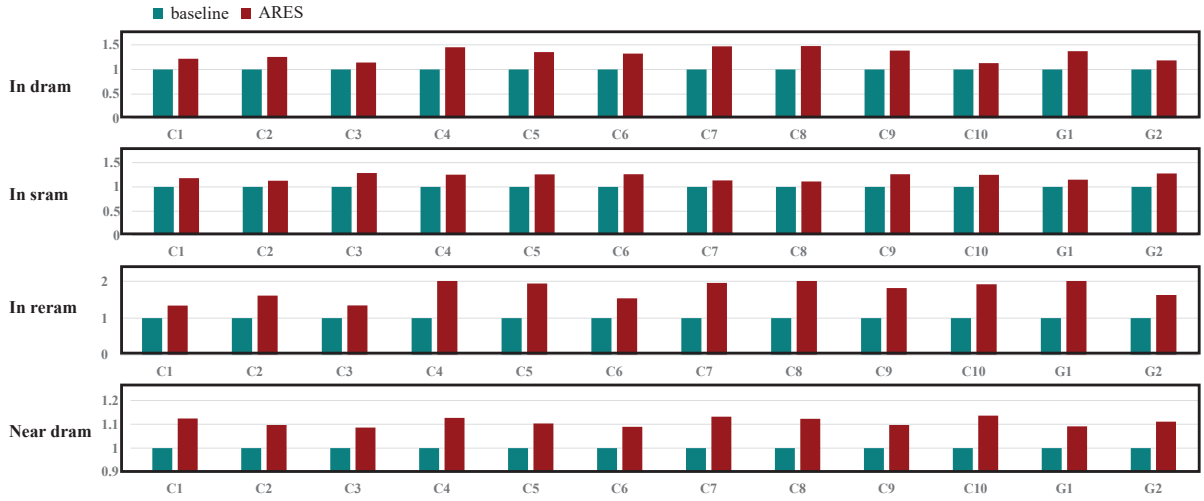


Fig. 5: Performance of single operator on PIM hardware

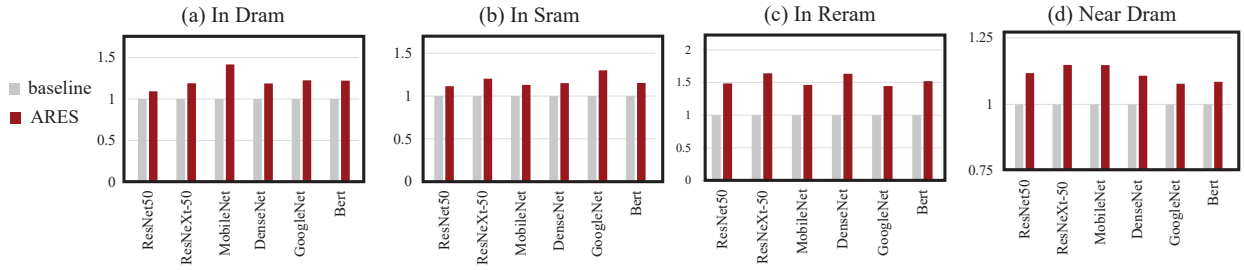


Fig. 6: End-to-end performance of full network compared with baseline

suitable for the MVM-ReRAM platform. We adopt the layer pipeline execution mode, and divide the crossbar resources among multiple network layers in order to make the overall computation time shortest.

Figure 6 shows the performance of the full networks. The overall performance improvement is up to 52%. Similar to the logic-SRAM and logic-DRAM platform, the performance improvements varies greatly across networks.

D. Results of PNMs

1) Single Operator Results: Figure 5 shows the performance results of our method and baselines. The performance improvements are 11.98% on average for PNM. This improvement is mainly from the reuse of memory resources. Because baseline does not analyze data reuse between compute instances, a PIM unit can store fewer data. This leads to the fact that the PIM unit needs to interact with other memory more frequently to obtain data, which increases the data transfer overhead. In the experimental results, we can find that the performance gap between different network layers is not large. Because they can all make full use of the computational resources, the difference mainly lies in the number of visits to the memory.

2) Full Network Results: Figure 6 shows the performance of the full networks. For entire networks, the performance improvements are 11.01% on average. Which is very similar to the performance the a single layer.

VII. Conclusion

In this paper, we have analyzed numerous PIM devices and developed a general abstraction for PIM hardware. This general abstraction includes two aspects: abstraction of compute functions and abstraction of memory mapping. With these abstractions, we have constructed a complete mapping space for PIM hardware and implemented an automated process to complete the mapping. We conducted experiments with commonly used neural network operators on four different hardware platforms. Compared with the previous template-based method, we achieved up to 70% performance improvement for a single operator and 50% performance improvement for the whole network.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China (NSFC) under grant No. U21B2017.

REFERENCES

[1] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy *et al.*, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *ASPLOS*, 2019.

[2] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *ISCA*, 2016.

[3] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *ISCA*, 2018.

[4] A. Biswas and A. P. Chandrakasan, "Conv-ram: An energy-efficient sram with embedded convolution computation for low-power cnn-based machine learning applications," in *ISSCC*, 2018.

[5] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *MICRO*, 2017.

[6] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," *ACM SIGARCH Computer Architecture News*, 2016.

[7] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin *et al.*, "Hardware architecture and software stack for pim based on commercial dram technology: Industrial product," in *ISCA*, 2021.

[8] A. Devic, S. B. Rai, A. Sivasubramaniam, A. Akel, S. Eilert, and J. Eno, "To pim or not for emerging general purpose processing in ddr memory systems," in *ISCA*, 2022.

[9] L. Lu, N. Guan, Y. Wang, L. Jia, Z. Luo, J. Yin, J. Cong, and Y. Liang, "TENET: A framework for modeling tensor dataflow based on relation-centric notation," in *ISCA*, 2021.

[10] L. Jia, Z. Luo, L. Lu, and Y. Liang, "Tensorlib: A spatial accelerator generation framework for tensor algebra," in *DAC*, 2021.

[11] S. Zheng, S. Chen, P. Song, R. Chen, X. Li, S. Yan, D. Lin, J. Leng, and Y. Liang, "Chimera: An analytical optimizing framework for effective compute-intensive operators fusion," in *HPCA*, 2023.

[12] S. Zheng, R. Chen, A. Wei, Y. Jin, Q. Han, L. Lu, B. Wu, X. Li, S. Yan, and Y. Liang, "Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction," in *ISCA*, 2022.

[13] S. Zheng, S. Chen, and Y. Liang, "Memory and computation coordinated mapping of dnns onto complex heterogeneous soc," in *DAC*, 2023.

[14] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, "Simdram: a framework for bit-serial simd processing using dram," in *ASPLOS*, 2021.

[15] M. Zhou, G. Chen, M. Imani, S. Gupta, W. Zhang, and T. Rosing, "Pim-dl: Boosting dnn inference on digital processing in-memory architectures via data layout optimizations," in *PACT*, 2021.

[16] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "Computedram: In-memory compute using off-the-shelf drams," in *MICRO*, 2019.

[17] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *MICRO*, 2017.

[18] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *ISCA*, 2019.

[19] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2014.

[20] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "'memristive'switches enable 'stateful'logic operations via material implication," *Nature*, 2010.

[21] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for rram-based in-memory computing using majority-inverter graphs," in *DATE*, 2016.

[22] R. B. Hur, N. Wald, N. Talati, and S. Kvatinsky, "Simple magic: Synthesis and in-memory mapping of logic execution for memristor-aided logic," in *ICCAD*, 2017.

[23] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *HPCA*, 2017.

[24] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee *et al.*, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *ISCA*, 2020.

[25] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, and Y. Xie, "Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator," in *HPCA*, 2021.

[26] G. Dai, Z. Zhu, T. Fu, C. Wei, B. Wang, X. Li, Y. Xie, H. Yang, and Y. Wang, "Dimmining: pruning-efficient and parallel graph mining on near-memory-computing," in *ISCA*, 2022.

[27] Y. Kwon, Y. Lee, and M. Rhu, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *MICRO*, 2019.

[28] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng *et al.*, "Conda: Efficient cache coherence support for near-data accelerators," in *ISCA*, 2019.

[29] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O'Halloran, D. Chen, J. Xiong *et al.*, "Application-transparent near-memory processing architecture with memory channel network," in *MICRO*, 2018.

[30] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. Vijaykumar, "Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning," in *MICRO*, 2020.

[31] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *MICRO*, 2018.

[32] M. Lenjani, A. Ahmed, M. Stan, and K. Skadron, "Gearbox: A case for supporting accumulation dispatching and hybrid partitioning in pim-based accelerators," in *ISCA*, 2022.

[33] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *ASPLOS*, 2017.

[34] "DDR4 SDRAM," https://www.micron.com/products/dram/ddr4-sdram.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[36] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *CVPR*, 2017.

[37] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[39] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015.

[40] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.