# PIMCOMP: An End-to-End DNN Compiler for Processing-In-Memory Accelerators

Xiaotian Sun, Xinyu Wang, Wanqian Li, Yinhe Han, Member, IEEE, and Xiaoming Chen, Member, IEEE

Abstract—In the past decade, various processing-in-memory (PIM) accelerators based on various devices, micro-architectures, and interfaces have been proposed to accelerate deep neural networks (DNNs). How to deploy DNNs onto PIM-based accelerators is the key to explore PIM's high performance and energy efficiency. The scale of DNN models, the diversity of PIM accelerators, and the complexity of deployment are far beyond the human deployment capability. Hence, an automatic deployment methodology is indispensable. In this work, we propose PIMCOMP, an end-to-end DNN compiler tailored for PIM accelerators, achieving efficient deployment of DNN models on PIM hardware. PIMCOMP can adapt to various PIM architectures by using an abstract configurable PIM accelerator template with a set of pseudo-instructions, which is a high-level abstraction of the hardware's fundamental functionalities. Through a generic multi-level optimization framework, PIMCOMP realizes an endto-end conversion from a high-level DNN description to pseudoinstructions, which can be further converted to specific hardware intrinsics/primitives. The compilation addresses two critical issues in PIM-accelerated inference from a system perspective: resource utilization and dataflow scheduling. PIMCOMP adopts a flexible unfolding format to reshape and partition convolutional layers, adopts a weight-layout guided computation-storage-mapping approach to enhance resource utilization, and balances the system's computation, memory access, and communication characteristics. For dataflow scheduling, we design two scheduling algorithms with different inter-layer pipeline granularities to support varying application scenarios while ensuring high computational parallelism. Experiments demonstrate that PIMCOMP improves throughput, latency, and energy efficiency across various architectures. PIMCOMP is open-sourced at https://github.com/ sunxt99/PIMCOMP-NN.

Index Terms—Processing-in-memory accelerator, deep neural network, end-to-end compiler, system-level optimization

#### I. INTRODUCTION

EEP neural networks (DNNs), with their powerful feature extraction and classification abilities, can excellently perform various intelligent tasks. According to neural scaling laws, the size and data of neural network models are continuously increasing to achieve better performance. Researchers have proposed a series of special-purpose accelerators (e.g., [1], [2]) to accommodate the models and speed

This work was supported in part by Strategic Priority Research Program of CAS under Grant XDB44000000, in part by National Natural Science Foundation of China under Grant 62122076, Grant 62025404, and Grant 62488101, in part by Key Research Program of Frontier Sciences, CAS under Grant ZDBS-LY-JSC012, and in part by Youth Innovation Promotion Association CAS. (Corresponding author: X. Chen)

X. Sun, X. Wang, W. Li, Y. Han, and X. Chen are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the University of Chinese Academy of Sciences, Beijing 100190, China (e-mail: sunxiaotian21s@ict.ac.cn; wangxinyu22s@ict.ac.cn; liwanqian20s@ict.ac.cn; yinhes@ict.ac.cn; chenxiaoming@ict.ac.cn).

up the inference process to cope with the ever-expanding neural networks. However, these accelerators are facing the memory wall challenge [3] as they suffer from high-cost data movement between memory and processing elements and encounter obstacles in enhancing energy efficiency.

Processing-in-memory (PIM) is expected to overcome the memory wall challenge as it binds data and computation, thereby circumventing the need for data movement. Among various PIM implementations, the resistive random-access memory (RRAM) has high density, low latency, and is easy to integrate with the CMOS technology [4], offering a broad application prospect. RRAM cells are usually integrated into a crossbar array, which can perform a matrix-vector multiplication (MVM) in O(1) time. The weights are programmed to be the conductances of the cells. The input activations are converted to voltages by digital-to-analog converters (DACs). According to the Kirchhoff's law, the output currents reflect the product of the weights (a matrix) and the inputs (a vector). Since the crossbar array operates in the analog domain, peripheral circuits such as DACs and analog-to-digital converters (ADCs) are needed to convert the signals. Due to the high memory density, parallel in-situ computing properties, elimination of weight movement, and the crossbar array formed by PIM devices can potentially meet the storage and computation requirements of DNNs. Crossbar arrays can also be constructed with other devices, such as ferroelectric field-effect transistors, magnetic random-access memories, phase-change memories, and even volatile static random-access memories. Based on this principle, previous works (e.g., [5]–[10]) have designed a series of crossbar array-based DNN accelerators.

These accelerators contain thousands of or more crossbar arrays, which challenge the deployment of DNN models due to the vast hardware scale. In addition, DNN models with different sizes and topologies demand meticulous attention for resource utilization, data scheduling, and memory optimization during deployment. Due to the different micro-architectures of various accelerators, it is uneconomical and unrealistic to manually design the deployment schemes for each DNN model on each accelerator as in previous works [5]–[10]. Therefore, a compiler that can adapt to various PIM architectures and automatically complete DNN model deployment is indispensable to improve the usability of PIM accelerators, which also helps build a PIM ecosystem [11].

To bridge DNN models and PIM accelerators, the compiler needs to be designed by considering the following aspects to make it universal, flexible, and efficient.

 For hardware: various PIM-based DNN accelerators have emerged, and it is cumbersome to design a specific

- compiler for each accelerator. Therefore, the compiler needs to be built on a high-level hardware abstraction, which can be lowered to specific PIM architectures. In addition, the compiler needs to be flexible and configurable to facilitate users quickly deploying DNNs on PIM accelerators with different scales of hardware resources.
- For software: the compiler should support a variety of DNN workloads. Besides, it should apply to different application scenarios, such as meeting low-latency or high-throughput requirements. Above all, the compiler should automatically complete the deployment, that is, transparently perform model reading, weight mapping, and output collecting without user intervention.
- System-level optimization: the compiler needs to handle resource allocation and dataflow scheduling effectively to unleash the hardware potential. For resource allocation, the compiler should make full use of the PIM resources to boost performance while balancing computing, memory access, and communication. For dataflow scheduling, the compiler should rapidly generate instruction streams for different scenarios and optimize the system performance bottlenecks. In addition, the compiler ought to have a profiler that provides an effective evaluation to steer the iterative performance optimization.

To address the challenges faced by PIM accelerators during DNN deployment, this paper proposes *PIMCOMP*, an end-to-end DNN compiler for PIM accelerators. A previous version of this work was published in [12], which provides an optimization scheme for resource allocation, task mapping, and pipeline dataflow through four stages: layer partitioning, weight replication, core mapping, and dataflow scheduling. The previous work offers a methodology that focuses on some deployment-specific issues, but falls short of meeting the compiler requirements stated above, as it lacks support for end-to-end compilation, integration of different compilation stages, and system-level optimizations. Nevertheless, the previous work lays out a design blueprint and algorithmic guidance for this work.

In this paper, we first review and contrast the existing PIM compilers. Then, we present the overall architecture and implementation details of *PIMCOMP*. *PIMCOMP* is a practical compiler that builds on Ref. [12] and enhances hardware compatibility, software support, and holistic performance. The new contributions of this paper are summarized as follows.

- We propose an end-to-end DNN compiler tailored for PIM accelerators, encompassing the frontend, optimizer, and backend, capable of inferring an entire DNN model. We design two pipelines to accommodate diverse application scenarios. We build a profiler to accomplish a comprehensive performance assessment for iterative compilation space optimizations.
- 2) We present a PIM accelerator abstraction adaptable to various academic accelerator designs. The accelerator abstraction comprises a hardware template with rich configurability, a set of pseudo-instructions fully abstracting the hardware's fundamental functionalities, and userspecified hardware execution patterns.

- 3) We introduce three-stage optimizations to complete the deployment of DNNs on PIM accelerators. In the layer partitioning stage, we propose array groups as the basic programming unit and utilize a flexible unfolding format to meet various resource demands. In the layout-computation mapping stage, we propose a weight-layout guided computation-storage-mapping method, utilizing genetic algorithms to optimize weight replication and layout and adaptively allocating computational tasks. In the dataflow scheduling stage, we propose scheduling algorithms maintaining high parallelism tailored to two pipelines.
- 4) We evaluate PIMCOMP's end-to-end deployment capability on three different architectures. The experimental results show that PIMCOMP achieves promising improvements in throughput, latency, and energy consumption compared with previous works.

#### II. COMPILATION TOOLS FOR PIM

Previous works have explored the end-to-end conversion from high-level programs to low-level code [13]–[15], incorporating optimizations during the progressive lowering. However, these approaches designed for CPUs and GPUs with separate storage and computation are not well-suited to PIM, due to its characteristic of integrating storage and computation. Furthermore, the high parallelism enabled by the PIM crossbar array reduces the effectiveness of conventional optimization strategies, which are tailored for multi-level loops.

Consequently, specialized compilation tools for PIM have been developed [16]–[23]. Among them, *Polyhedral* [16], *Co-Design* [17], *SongC* [18], and *PIMCOMP* focus on DNNs, while *TC-CIM* [19], *TDO-CIM* [20], *OCC* [21], and *PUMA* [22] target a broader range of machine learning algorithms. In addition to these, *CINM* [23] supports more general applications, such as time series analysis. With DNNs as the target application, Table I summarizes these compilation tools from the following perspectives.

(1) **Design philosophy**: The primary concept of PIM is to bind computation and data, namely, performing computations where data reside. Thus, the data layout is crucial. For accelerating DNNs on PIM, there is a crucial strategy called weight replication [5], [7], [24]. By replicating a layer's weights by R times, the computational parallelism of that layer is increased by R times so the performance can be boosted by R times at most. TC-CIM [19], TDO-CIM [20], CINM [23], OCC [21], Polyhedral [16], Co-Design [17], and PUMA [22] do not consider the interaction between weight replication and weight layout, and simply map computational tasks to PIM arrays. This approach may lead to suboptimal performance and resource under-utilization. Song C [18] considers a storage-computation-disentangled approach, which resembles non-PIM accelerators, resulting in additional overhead due to weight movement. PIMCOMP employs a weight-layout guided computation-storage-mapping method, which carefully determines the weight layout for all layers under weight replication consideration and subsequently adaptively maps computational tasks to PIM arrays based on the weight replication and layout. This approach not only fully utilizes hardware

E2E for DNN | HW config. Design philosophy Sche. gran. App. diver. Sys. opt. TC-CIM [19] Naive computation-storage-binding Layer-wise MVM oper. × Low Low TDO-CIM [20] Layer-wise MVM oper. X Naive computation-storage-binding Low Low CINM [23] Naive computation-storage-binding Layer-wise MVM oper. × High Low  $\mathbf{P}^{\mathrm{I}}$ × Р OCC [21] Naive computation-storage-binding Layer-wise MVM oper. Low Low Layer-wise MVM oper. X P/H<sup>2</sup> Polyhedral [16] Naive computation-storage-binding Medium Medium Co-Design [17] Naive computation-storage-binding CG-based MVM oper. Medium P/H Medium **PUMA** [22] Naive computation-storage-binding CG-based MVM oper. Medium Medium P/M<sup>2</sup> Layer-wise MVM oper. SongC [18] P/M Storage-computation-disentangled Medium Low **PIMCOMP** Weight-layout guided computation-storage-mapping CG-based Conv. oper. High High P/H/M

TABLE I: Summary and comparison of different compiling tools for PIM, focusing on DNN applications.

resources, but more importantly, eliminates mismatch between the requirements of computation and storage.

(2) Schedule granularity: TC-CIM [19], TDO-CIM [20], CINM [23], OCC [21], Polyhedral [16], and SongC [18] convert convolution layers into fundamental MVM operators in a layer-by-layer manner. This is a natural idea, but, such a coarse-grained granularity presents a challenge: when multiple layers are mapped to a core, they need to be computed sequentially, leading to resource under-utilization. On the contrary, in Co-Design [17] and PUMA [22], scheduling is performed at the granularity of MVM operators across the model, incurring significant overhead in parallelism analysis and operation linearization, resulting in prolonged compilation time. To fully exploit hardware resources and facilitate efficient scheduling, PIMCOMP adopts a convolution operator<sup>1</sup>-level scheduling strategy, ensuring that operators without structure and data conflicts can be executed in parallel, thereby enhancing operator-level parallelism.

(3) **End-to-end for DNN**: An end-to-end DNN compiler should act as a black box that compiles a user-given DNN model into logically correct instructions or operations. Two key features should be realized for an end-to-end compiler. First, the compiler should not ask users to rewrite the model. *OCC* [21], *CINM* [23], *Co-Design* [17], *SongC* [18], and *PIMCOMP* eliminate the need for rewriting the models, while the others require users to manually convert the model into the special input format that the compiler can accept. Second, the compiler should have the capability to compile an entire DNN model. *TC-CIM* [19], *TDO-CIM* [20], *CINM* [23], *OCC* [21], and *SongC* [18] can only compile single layers or blocks within the networks. *Co-Design* [17] and *PIMCOMP* both use the ONNX format<sup>2</sup> as input to avoid rewriting and provide mechanisms to support compilation of the entire DNN model.

(4) Hardware configurability: TC-CIM [19], TDO-CIM [20], and OCC [21] assume relatively fixed hardware structures, which limits the hardware configurability. CINM [23] provides hardware abstraction, enabling support for various hardware platforms. The other compilers support flexible hardware structures. Among them, PIMCOMP supports a rich set of configurable parameters of various hardware components, such as the scale of cores, the size of

PIM crossbar arrays, and bandwidths of different levels of memories and network-on-chip (NoC).

(5) **Application-scenario diversity:** Real-world tasks have different performance requirements. For example, cloud devices need high throughput, while mobile devices need low latency. Since PIM accelerators store all weights in crossbar arrays, compilers can adjust the inter-layer pipeline granularities to suit various application scenarios. *TC-CIM* [19], *TDO-CIM* [20], *CINM* [23], *OCC* [21], and *SongC* [18] can only perform layer-by-layer deployment for a single sample. *Polyhedral* [16] and *CO-Design* [17] implement samplegranularity pipelines to adapt to high-throughput scenarios. *PUMA* [22] designs an inter-core pipeline at the MVM granularity to reduce latency. *PIMCOMP* designs specifically optimized scheduling algorithms for two different scenarios, enhancing the applicability of PIM hardware.

(6) System optimization: To fully exploit the capability of PIM hardware, the compiler should explore various optimization opportunities, such as performance, hardware resource utilization, and memory orchestration. TC-CIM [19] and TDO-CIM [20] identify and offload suitable operations to PIM hardware but lack optimization measures. CINM [23] and OCC [21] apply loop unrolling on this basis to enhance the parallelism. Polyhedral [16] adopts a polyhedral model to optimize inference performance and increases the resource utilization by weight replication. Co-Design [17] and PUMA [22] employ graph partitioning algorithms to minimize data transmission between cores. For memory orchestration, *PUMA* [22] reuses memory space and SongC [18] compresses redundant memory blocks. PIMCOMP leverages resources and boosts performance through automated layout-computation mapping (Section VI) and devises runtime memory management strategies for low-latency pipelining (Section VII).

#### III. PIM ACCELERATOR ABSTRACTION

To make *PIMCOMP* compatible with various PIM accelerators, we propose a high-level abstraction that includes an architecture template, pseudo-instructions as a software-hardware interface, and configurable execution patterns. This abstraction acts as a hardware abstraction layer [25], isolating hardware differences from the compiler to improve *PIMCOMP*'s compatibility and portability. In this section, we first introduce the architecture template, which macroscopically defines the functionality and organization of the fundamental modules. Then, we present the software-hardware interface serving as the interaction layer between DNN models and the abstract

<sup>&</sup>lt;sup>1</sup>P: performance. <sup>2</sup>H: hardware resource utilization. <sup>3</sup>M: memory orchestration

<sup>&</sup>lt;sup>1</sup>A convolution operator is the computation associated with a sliding window on the input feature maps and all filters, producing an output element on each output feature map.

<sup>&</sup>lt;sup>2</sup>https://onnx.ai.

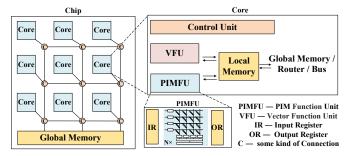


Fig. 1: Architecture template.

architecture. Based on this, we propose configurable execution patterns to specify the execution logic of the abstract hardware, facilitating the compilation adapting to the actual hardware execution patterns.

#### A. Architecture Template

The architecture template is a representative multi-level configurable and scalable architecture derived from previous PIM accelerators [5], [22], [22], [26]. Fig. 1 illustrates the architecture template comprising chips, cores, PIM function units (PIMFUs), crossbar arrays, and multi-level storage. Given our commitment to expanding the applicability of *PIMCOMP*, this template does not introduce additional innovations in hardware design. Its distinguishing feature lies in the extensive configurability at various levels. The configurable parameters, listed in Table II, facilitate the instantiation of the template into a tailored accelerator. The **accelerator** architecture consists of multiple chips interconnected by off-chip connections, such as PCIe.

Each **chip** contains multiple cores connected to the global memory. These cores are interconnected by configurable methods, such as buses, NoCs, or shared memories. The cores perform parallel computation and can communicate by synchronous or asynchronous mechanisms.

The **core** is the computing unit that performs MVMs with a PIMFU and vector computations with vector function units (VFUs), with a local scratchpad memory to store intermediate results. Users can customize the VFU to perform different vector operations, such as result accumulation with an adder tree [27], max pooling with max pool units [5], activation with activation units [5], or various vector operations with integrated multi-function vector units [22]. The local memory represents an abstraction of all storage units within a core, facilitating data exchange with both PIMFUs and VFUs while also storing data transmitted between cores. Moreover, the local memory can handle padding, concatenation, and splitting operations. The presence of the local memory enables the compiler to schedule the dataflow, thereby achieving an on-chip pipeline flexibly.

The **PIMFU** is a collection of PIM crossbar arrays and their peripheral circuits intended to process MVMs. Due to the limited precision of crossbar array cells, high-precision weights usually need multiple crossbar arrays to store collaboratively [27]. Similarly, a signed weight may need two crossbar arrays to store the positive and negative parts separately [6].

TABLE II: Configurable parameters for the architecture template.

Level	Parameter	Level	Parameter
	#Chip in X		VFU functionality
Accelerator	#Chip in Y		#VFU
	Off-chip bandwidth	Core	LocalMem bandwidth
	#Core in X		LocalMem size
	#Core in Y		Instruction execution model
	Connection type		#Crossbar in X
Chip	Connection bandwidth	PIMFU	#Crossbar in Y
	Comm. mechanism		Management granularity
	GlobalMem bandwidth	Crossbar	
	GlobalMem size	Array	Cell precision

To support multi-precision deployment [28] and simplify the compiler design, we use a logical array with each cell storing a signed weight with arbitrary precision to replace multiple physical arrays that form the same weight jointly in the optimizer stage. The compiler records the mapping relationships between the logical and physical arrays and performs weight-bit splitting in the backend stage based on this information. Moreover, the arrays are loosely coupled in PIMFU, meaning that their organization is not specified. To adapt to different control granularities of various PIM accelerators, we design a management granularity to represent the basic control unit of PIMFU, which ranges from fine to coarse, including a single array, a specific number of arrays, and all arrays in PIMFU.

Users can configure the size and cell precision of the **crossbar array** in PIMFU. By configuring different array sizes for different cores, a mixed-size deployment can be realized [29].

# B. Software-Hardware Interface of Pseudo-Instructions

The software-hardware interface is exposed at the core level of the hardware template because the core is responsible for managing and offloading computation, memory access, and data transmission tasks. By programming each core, the compiler can drive the entire accelerator for DNN inference.

We design a set of pseudo-instructions as the software-hardware interface as listed in Table III, where each pseudo-instruction abstracts a fundamental functionality of the core. For example, mvm controls the array group (the concept of array group will be introduced in Section V) identified by %idx to perform an MVM operation. Within vec, the %op represents the identification of various vector operations, such as activation operations of a single vector, arithmetic operations between two vectors, and comparative operations, among others.

Previous studies have suggested different application program interfaces (APIs) or instruction set architectures (ISAs) for PIM accelerators, but they are significantly different from the interface presented in this paper. *TDO-CIM* [20] develops a PIM runtime library to invoke the PIM co-processor. However, its management only covers data transmission between the host and device and the launch of the general matrix multiplication (GEMM) kernel. This coarse-grained management fails to exploit the full potential of PIM hardware and is inadequate for achieving sophisticated scheduling, such as fine-grained interlayer pipelines and data reuse across convolution operators.

Operation type	Pseudo-instruction	Target	Description		
	mvm(%idx, %dst, %src,	PIMFU	Array group with index %idx reads data of length %len at %src in LocalMem		
Computation	%len)	1 IIVII O	performs MVM, and then writes the result back to LocalMem at %dst.		
	vec(%op, %dst, %src1,	VFU	VFU performs vector calculations on data of length %len at %src1 and %src2		
	%src2, %len)	VITO	in LocalMem, and writes the result to %dst. %op identifies the operations.		
	copy(%dst, %src, %len)	LocalMem	Copy the data at %src of length %len to %dst in LocalMem		
Memory	write(%dst, %len, %imm) LocalMem		Write %imm to data of length %len at %dst in LocalMem		
Wichiory	load(%dst, %src, %len)	GlobalMem→LocalMem	Copy data of length %len from %src in GloalMem to %dst in LocalMem		
	store(%dst, %src, %len)	LocalMem→GlobalMem	Write data of length %len from %src in LocalMem to %dst in GlobalMem		
Communication	send(%idx, %src, %len)	Core → Core	Transmit data of length %len from %src in LocalMem to Core %idx		
Communication	recv(%idx, %dst, %len)	$Core \rightarrow Core$	Receive data from Core %idx and write to LocalMem at %dst of length %len		

TABLE III: Software-hardware interface of pseudo-instructions.

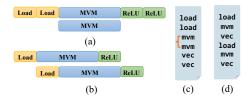


Fig. 2: Example of adjustment based on instruction execution model. (a) and (c) depict the computational schematic and compilation results for in-order instruction execution model, respectively. (b) and (d) represent the computational schematic and compilation results for out-of-oder instruction execution model, respectively.

CINM [23], OCC [21], and Polyhedral [16] face the same issue. Co-Design [17] and PUMA [22] design their ISAs for the programmable hardware they propose. However, their ISAs are only compatible with specific architectures, lacking the generality and universality required for broader applicability.

#### C. Execution Patterns

Some configurable parameters of the hardware template pertain to the execution characteristics of the accelerator. To ensure the compilation results match the underlying hardware's operational logic, *PIMCOMP* appropriately adjusts and optimizes the pseudo-instruction order based on the user-specified hardware execution patterns. We illustrate *PIMCOMP*'s efforts using the instruction execution model and communication mechanism in Table II as examples.

- (1) Instruction execution model: For accelerators targeted by *Co-Design* [17] and *PUMA* [22], capable of in-order instruction execution, each computation entails driving all crossbar arrays within the core to enhance computational parallelism. The computational schematic is illustrated in Fig. 2(a), resulting in the compilation outcome depicted in Fig. 2(c), where inputs for each array are loaded, followed by simultaneous computation across all crossbar arrays. In contrast, accelerators featuring out-of-order instruction execution can operate as depicted in Fig. 2(b). Consequently, the compilation outcome, as shown in Fig. 2(d), achieves overlapping of memory access and computation.
- (2) Communication mechanism: For accelerators employing an asynchronous communication mechanism, typically only a send instruction is required, with recv implicitly completed by hardware interrupts or other signaling mechanisms. *PIMCOMP* needs to record the address and length

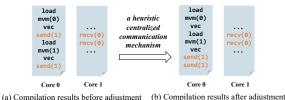


Fig. 3: Example of adjustment based on communication mechanism. The arguments of mvm, send, and recv are the value of %idx.

allocated by the compiler for data reception. For accelerators that adopt a synchronous communication mechanism, the compiler ensures that send and recv are paired to maintain the legitimacy of each inter-core communication. Since blocking occurs with each communication, we reduce synchronization overhead and enhance system parallelism by a heuristic centralized communication mechanism, adjusting the order of send and recv. For illustration, Fig. 3(a) shows the original compilation result, where the core waits for the completion of the first send(1) instruction, thereby hindering other crossbar arrays from performing MVM operations. Therefore, in Fig. 3(b), we adjust the position of the send(1) instruction to ensure the independent crossbar arrays complete the computational tasks in parallel, followed by the data transmission.

#### IV. PIMCOMP OVERVIEW

Fig. 4 illustrates the overview of *PIMCOMP*, which consists of four main components: a frontend, an optimizer, a backend, and a profiler. The user inputs the DNN model and the hardware configuration, and *PIMCOMP* automatically performs the deployment task. The **frontend** reads the DNN model and extracts the weight data and the structure intermediate representation (IR). Simultaneously, the frontend configures the optimizer and profiler based on the hardware configuration. In the **optimizer** stage, PIM-specific strategies are employed at multiple levels to enhance resource utilization and perform dataflow scheduling, producing weight mapping information and pseudo-instruction streams. In the backend stage, the weights are quantized and mapped according to the compiler outcomes, and the pseudo-instruction streams are translated into the actual hardware operation streams using the hardware library provided by the user. The **profiler** provides the performance metrics for the iterative optimization of the optimizer.

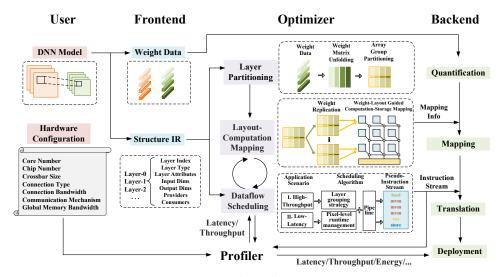


Fig. 4: Overview of PIMCOMP.

#### A. Frontend

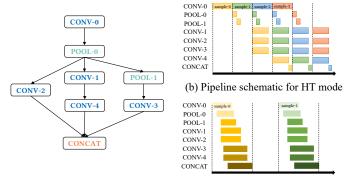
The frontend parses the DNN model and converts it into IRs, eliminating the need for users to rewrite the model. *PIMCOMP* supports DNN models described in the ONNX format, which can be easily obtained from various frameworks such as PyTorch and TensorFlow, thus enabling *PIMCOMP*'s compatibility with multiple deep learning frameworks.

To enable efficient deployment, the frontend adopts a structure-weight splitting strategy, which parses the DNN model and produces separated structure IR and weight data. The structure IR contains the topology of the network and the parameters of the layers, which act as a lightweight proxy for the DNN model, facilitating fast iterative optimization in the optimizer stage. Moreover, the frontend performs graph-level optimizations such as layer fusion and elimination. The weight data consists of the actual weight values of each layer, which will be mapped to the physical hardware in the backend stage. Additionally, the frontend initializes the compiler and profiler based on the hardware configuration information.

#### B. Optimizer

The optimizer is the core component of *PIMCOMP*. We adopt a multi-level compilation approach focusing on three general stages to address the challenges associated with resource utilization and dataflow scheduling. Layer partitioning (Section V) specifies the compilation granularity for the entire compilation after unfolding the weight data into the matrix. Layout-computation mapping (Section VI) delineates the layout of weight data and the allocation of computational tasks among the PIM arrays considering weight replication. Dataflow scheduling (Section VII) schedules the dataflow and constructs a pseudo-instruction stream for each core. The latter two stages form a closed-loop iterative optimization to boost inference performance under the performance feedback by the profiler.

To enhance the versatility of the compiler, *PIMCOMP* has devised two compilation modes characterized by interlayer pipelining: high-throughput (HT) and low-latency (LL),



(a) A typical neural network block (c) Pipeline schematic for LL mode

Fig. 5: Pipeline schematic under HT and LL modes.

as detailed in Fig. 5. The fundamental difference between them lies in the granularity of the inter-layer pipeline. In the HT mode, the DNN infers layer by layer, with different layers processing data from different samples. Inter-layer data transmission occurs after the computation of the current layer is complete, allowing for a higher degree of parallelism in computation execution, as computation is infrequently interrupted by communication. In the LL mode, as soon as a layer calculates an output, it passes it to the subsequent layer. As long as a layer receives enough data to perform computations, it can calculate promptly, thereby reducing inference latency.

#### C. Backend

The backend stage is responsible for interfacing *PIMCOMP* with the actual hardware. As the architecture abstraction obscures some specific hardware details, the backend stage refines and supplements them. Initially, the backend quantifies the weight data for each layer. Subsequently, it partitions the weight data, performs bit splitting, and then programs it into the actual PIM arrays based on the weight mapping information obtained from the optimizer. Thereafter, the backend translates the pseudo-instruction stream into hardware

primitives, leveraging the hardware library provided by the user, which enables the physical accelerator to autonomously execute deployment tasks. *PIMCOMP* leaves an interface for the backend so that users can provide the backend for the specific hardware.

# D. Profiler

The profiler provides performance metrics for the optimizer stage and pre-deployment evaluation for the accelerator. The profiler utilizes the architectural parameters provided by the user in Table II and the pseudo-instruction stream to conduct performance and functional simulations of the accelerator. In terms of latency, by establishing timing axes for multiple components, the profiler can simulate intra-core instructionlevel parallel execution scenarios, taking into account the structural and data conflicts, as well as the synchronization overhead of inter-core communication. Additionally, if the user supplies power data for each component, the profiler can synthesize pseudo-instructions, establish the relationship between pseudo-instructions and components, and obtain the inference power dissipation. The profiler also collects statistical data such as resource utilization, memory footprint, inter-core data transmission, and memory access volume to inquire into the processing details.

#### V. LAYER PARTITIONING

Facing accelerators containing thousands of PIM arrays, organizing and managing the numerous resources becomes a critical challenge for the programming model. Due to hardware array size limitations, the weight data of a layer is often distributed across multiple crossbar arrays. If the compiler uniformly manages all arrays within a layer, deployment flexibility is compromised. In cases where a core's PIM resources are insufficient to accommodate a layer, deployment may fail. On the other hand, if the compiler handles each array separately, the management flexibility is high. However, due to the exponential growth of the solution space, the toolchain's execution efficiency would be compromised, resulting in greater runtime overhead. To strike a balance between flexibility and efficiency, we propose the concept of array groups as the fundamental unit within the programming model. In this section, we delve into the division rules and advantages of array groups. Additionally, we introduce the flexible unfolding format, which leverages different transformation methods to unfold the weight data.

#### A. Array Group

Given the ability of PIM arrays to perform MVM operations in constant time, a common practice for accelerating DNNs on PIM involves unfolding the weight data of a convolutional layer into a set of parallel weight matrices, unfolding the sliding window of input feature maps into a set of input vectors, and facilitating PIM arrays to compute the multiplication of weight matrices and input vectors. Suppose we have a convolutional layer whose weight data is unfolded (the specific unfolding process is described in Section V-B) into

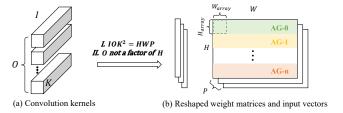


Fig. 6: Unfolding strategy for convolutional kernels and partitioning strategy for the set of reshaped weight matrix.

a set of P weight matrices of size  $H \times W$  each, and every sliding window is reshaped into P input vectors of length H each, as illustrated in Fig. 6(b). We partition every weight matrix vertically into several groups based on the height of the array  $H_{array}$  where each group, termed an array group (AG), contains  $\lceil W/W_{array} \rceil$  arrays. These resulting array groups serve as the fundamental units for mapping and scheduling in the programming model of PIMCOMP.

The rationales behind our design are as follows. From a hardware perspective, an AG physically corresponds to a set of crossbar arrays in a core. Since crossbar arrays within an AG share identical inputs, input data can be broadcasted to these arrays, thus avoiding redundant data transmissions and alleviating on-chip bandwidth. In terms of mapping flexibility, the existence of AGs decouples the rigid association between cores and layers. AGs from the same layer can be mapped to different cores, and a single core can accommodate AGs from distinct layers, allowing for flexible mapping of weight data and empowering compilers to explore a wide range of optimization possibilities. Regarding scheduling efficiency, arrays within the same AG can be managed by the same pseudoinstruction (mvm), enabling shared inputs and parallel outputs, thereby facilitating dataflow scheduling and pseudo-instruction generation. During the backend phase, pseudo-instructions at the granularity of AGs are transformed into primitives that match the accelerators' management granularity.

# B. Flexible Unfolding Format

Section V-A highlights that utilizing PIM arrays to accelerate DNNs necessitates reshaping weight data into a set of 2D weight matrices. It is observed that different unfolding strategies may exhibit distinct computational and memory access characteristics. Consequently, the compiler needs to support flexible unfolding formats to address various hardware resource constraints and performance requirements. Hereafter, we exemplify the characteristics of different unfolding methods using a convolutional layer with input feature map size  $F_{in}$ , output feature map size  $F_{out}$ , input channel number I, output channel number O, and convolutional kernel size K.

As illustrated in Fig. 6, each unfolding method can be represented by a tuple (H, W, P), where H and W denote the height and width of the weight matrix, respectively, and P represents the number of parallel weight matrices. Since the number of weight elements remains constant before and after unfolding, it follows that  $HWP = IOK^2$ . Additionally, as the direction of H requires data accumulation, we can not

TABLE IV: Theoretical performance of different weight unfolding methods.

Type (H,W,P)	Computation cycle	Load volume	Additional memory
$(I,O,K^2)$	$F_{out}^{2}$ (S)	$F_{in}F_{out}KI$ (M)	$K^2I + K^2O$ (L)
$(I, OK^2, 1)$		$F_{in}^2 I$ (S)	$I + K^2O$ (M)
(IK, O, K)	$F_{out}^{2}$ (S)	$F_{in}F_{out}KI$ (M)	$K^2I + KO$ (M)
(IK, OK, 1)	Fin Fout (M)	$F_{in}F_{out}KI$ (M)	KI + KO (S)
$(IK^2, O, 1)$	$F_{out}^2$ (S)	$F_{out}^2 K^2 I$ (L)	$K^2I + O$ (M)

choose H such that O is a factor of H according to the convolution computational rule. Any unfolding method that satisfies these two rules is considered valid. The most common unfolding method is  $(IK^2, O, 1)$  [7], [27], which unfolds the weight into a single matrix. To reuse input data and reduce loading overhead,  $(I, O, K^2)$  is adopted in [30], requiring further accumulation of results from  $K^2$  parallel weight matrices to obtain the complete result. Similarly, (IK, OK, 1) is utilized to achieve the same objective [31]. Furthermore, a reconfigurable architecture is introduced in [32] to support different unfolding strategies.

Considering resource utilization, we have selected five valid unfolding methods, as shown in Table IV. Besides, we outline the theoretical computation cycles, data loading volume, and additional memory requirements. Here, the data loading volume refers to the amount of input data retrieved from the global memory, while the additional memory requirement denotes the memory overhead needed for input reuse or intermediate result storage. Users can flexibly select the unfolding method based on hardware resources and optimization objectives. In this work, we employ a greedy method to determine the unfolding scheme: prioritizing minimal computation cycles while pursuing minimal data loading volume in the HT mode and minimal memory requirement in the LL mode.

# VI. LAYOUT-COMPUTATION MAPPING

During the DNN inference process, each AG undergoes numerous iterations of load-computation-transmission, which requires the comprehensive utilization of hardware computation, memory, and communication capabilities. The spatial distribution of AGs across the cores and the allocation of computational tasks directly influence these three system characteristics. Furthermore, weight replication is a crucial yet often overlooked aspect by previous works, as it enhances resource utilization and improves performance.

To adhere to the characteristics of PIM and fully exploit hardware performance, we propose the weight-layout guided computation-storage-mapping. For weight layout, the genetic algorithm (GA) is utilized to determine the weight replication and distribution of AGs throughout all the cores (Section VI-A). For computation-storage-mapping, we flexibly determine the allocation of computational tasks to AGs based on the weight layout for both pipeline modes (Section VI-B).

# A. GA-based Weight Layout Optimization

As the number of replicas in a layer increases, the amount of AGs it contains also rises, thus expanding the search space for AG layout. To explore the vast solution space fully, we propose

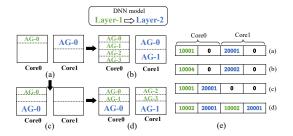
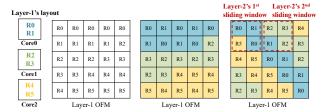


Fig. 7: Example of using GA to optimize weight layout considering weight replication. (a) Common practice. (b) Preferred approach of HT mode. (c) Intermediate result. (d) Preferred approach of LL mode. (e) Chromosome of (a)-(d).

employing GA to jointly optimize weight replication and AG layout. To represent the mapping relationship between all cores and AGs using a single chromosome, we adopt a value-position combined encoding method, where each gene's value represents the number of AGs, and its position signifies the mapped core. Specifically, we define the gene length for each core as  $max\_layer\_num\_in\_core$ , constraining the number of layers each core can accommodate. Consequently, the length of each chromosome is  $max\_layer\_num\_in\_core \times core\_num$ . The numerical value of each gene is an integer, calculated as  $layer\_index \times 10000 + AG\_num$  denoting  $AG\_num$  AGs of the layer with index  $layer\_index$ , with 10000 indicating the core can accommodate up to 9999 AGs for that layer. Fig. 7(e) illustrates the chromosomes for mappings Figs. 7(a)-(d) when  $max\_layer\_num\_in\_core = 2$ .

In the initialization phase, we set the replication factor of each layer to 1. Mutation is a critical phase for performance enhancement. We devise the following mutation strategies to explore the solution space composed of AG distribution and weight replication: I. Randomly select a gene and either increase or decrease its AG count. If the gene's value is 0 originally, indicating an unassigned position, we randomly assign a layer to it with a replication factor of 1. II. Randomly select two genes and partially swap their positions. Here, "partially" implies that each gene may only contribute a portion of AGs for exchange, and "swap" denotes placing the exchanged AGs into the zero-valued positions of the original gene's core. We ensure the legality of each chromosome throughout the mutation process. For the fitness function, our previous work [12] employs analytical models for evaluation. In this work, we assess using the profiler. Specifically, for each mapping represented by a chromosome, we perform the dataflow scheduling process mentioned later to generate a pseudo-instruction stream. This stream is not the final complete sequence; rather, it is a simplified version used for rapid evaluation by the profiler to obtain the performance considering computation, memory access, and communication as the fitness function.

Fig. 7 presents an example of using GA to optimize the weight layout for the HT and LL modes, where the PIM system includes two cores, and the DNN comprises two convolutional layers. Each layer has one AG, with the AG from Layer-1 and Layer-2 occupying 1/4 and 1/2 of the PIM



(a) Weight layout (b) Naive approach (c) Approach for HT mode (d) Approach for LL mode

Fig. 8: Different computational tasks allocation approaches.

resources within a core, respectively. Fig. 7(a) depicts the original method, where each core contains only one layer—a common approach in previous works [5], [7], [27]. This leads to suboptimal resource utilization and failure to leverage PIM's parallel computing advantages. For the HT mode, with modest inter-core communication, the key to improve performance lies in reducing memory access. The GA tends to find the mapping shown in Fig. 7(b), where Layer-1 and Layer-2 have replication factors of 4 and 2, respectively, with their AGs clustered. This approach maximizes resource utilization and enhances data reuse potential due to overlapping sliding windows of convolutional layers. For the LL mode, communication often becomes the bottleneck; hence, achieving a balance between computation and communication is the optimization direction. The GA tends to find the mapping shown in Fig. 7(d), where data produced by Layer-1 can directly serve Layer-2 within the same core, thereby reducing a portion of data communication.

# B. Adaptive Computation-Storage-Mapping

A convolutional layer exhibits local connectivity, thus its inference process involves multiple convolution operators, each performing computation between a sliding window and weight data. According to the computational features of PIM and the unfolding method employed by PIMCOMP, the computational task of each convolution operator is allocated to a replica of the convolutional layer. Fig. 8 illustrates an example of computational task allocation by *PIMCOMP*. Layer-1 possesses 6 replicas (denoted as R0 to R5 in Fig. 8(a)), each housing one AG, evenly distributed across Core-0 to Core-2. For illustrative purposes, in Figs. 8(b)(c)(d), we represent the computational task allocation for Layer-1 as the partitioning of the output feature map (OFM), where each point in the OFM represents the computational task of one convolution operator. Fig. 8(b) depicts the original method of evenly distributing computational tasks among the various replicas.

Fig. 8(c) shows the allocation approach in the HT mode. For each layer, the number of computational tasks undertaken by each core is proportional to the number of replicas of that layer housed within the core, ensuring a uniform distribution of computational tasks across all cores. Due to the characteristics of convolutional layers, there is typically an overlap between adjacent sliding windows in the input data, which is pre-prepared in the HT mode. Consequently, each replica alternately completes adjacent computational tasks within each core, enabling input data reuse and reducing global memory access. Although allocation in Fig. 8(b) also achieves input

reuse, when facing larger input feature maps, there is no overlap between the input windows of R0 and R1. Therefore, R0 and R1 need to separately maintain a portion of memory for data reuse. In contrast, in Fig. 8(c), R0 and R1 can share input data, significantly reducing memory requirements.

Fig. 8(d) depicts the allocation approach in the LL mode. To rapidly generate results for subsequent layers, we utilize complete computational parallelism to execute computational tasks sequentially. This allocation approach will increase data communication to some extent. Therefore, in Section VII, we will introduce a pixel-level transmission and memory allocation mechanism designed for the LL mode to reduce communication overhead while reducing memory requirements. For non-convolutional and non-fully-connected layers, which utilize the VFUs within the core, our allocation criterion is to minimize data communication greedily. As indicated by the red dashed box in Fig. 8(d), assuming Layer-2 is a pooling layer, its first windowed operation is completed in Core-0 since three-quarters of the data needed by the first window is stored in Core-0, and the second windowed operation is completed in Core-1 likewise. Consequently, we replicate Layer-2 and distribute the computational tasks among multiple cores according to this rule.

#### VII. DATAFLOW SCHEDULING

In general, PIM accelerators store the weights in PIM cross-bar arrays, which also serve as computational units. Ideally, these arrays are capable of operating in parallel. The nature of PIM endow DNN models with the potential for concurrent computation across various layers. Consequently, this leads to the development of an on-chip pipeline suitable for PIM, enabling inter-layer network pipelining on the chip, without the need for multiple chips. To cater to different application scenarios, we design the HT and LL modes by adjusting the granularity of the pipeline. This section presents the optimization considerations for each mode and the corresponding scheduling algorithms that generate pseudo-instruction streams for each core based on the allocation result of computational tasks obtained in Section VI.

# A. High-Throughput Mode

In the HT mode, a DNN infers layer by layer, suitable for scenarios involving continuous input streams. Fig. 5(b) illustrates its primary characteristics: layers with data dependencies do not overlap in runtime within the same cycle. Consequently, each layer preserves its output via the global or local memory, transmitting it to its consumers upon the cycle's completion.

However, in scenarios like autonomous driving that demand high-throughput and real-time capabilities, the first-batch latency directly impacts decision speed and response time. To enhance system service quality, we propose a layer grouping strategy optimized for first-batch latency. It groups the layers of the original computational graph based on two principles: I. Group data-dependent layers without compromising throughput, and II. Group independent layers for concurrent execution. During scheduling, layer groups are executed in the topological order. Independent layers within a group can

#### **Algorithm 1:** High-throughput dataflow scheduling. Input: LayerGroupList 1 CompletedLayerList ← [] **while** CompletedLayerList.Len() != LayerNum **do** 2 foreach LayerGroup in LayerGroupList do 3 4 IndepLayerList $\leftarrow$ LayerGroup.Layers.IndependentLayers() foreach Layer in IndepLayerList do 5 if Layer.Op == CONV or FC then foreach RepLayer in Layer.ReplicationList if RepLayer.TaskIdx < 8 RepLayer.TaskNum then RepLayer.Run(RepLayer.TaskIdx) 10 RepLayer.TaskIdx += 1Layer. TaskIdx += 111 else 12 Layer.Run(Layer.TaskIdx, CoreNum) 13 Layer.TaskIdx += CoreNum if Layer.TaskIdx == Layer.TaskNum then 15 CompletedLayerList.Append(Layer.Idx) 16 LayerGroup.Layers ← 17 UpdateLayers(LayerGroup.Layers,

execute simultaneously, while data-dependent layers follow the topological sequence. These rules reduce the first-batch latency from the product of the number of layers and the cycle to the product of the number of layer groups and the cycle.

CompletedLayerList)

Taking Fig. 5(b) with 8 layers as an example, assume the time  $T_{CONV0}$  required to execute layer CONV-0 is the longest, dictating the pipeline's cycle. The first-batch latency would normally be  $8T_{CONV0}$ . We observe that the combined execution time of POOL-0 and POOL-1 is less than  $T_{CONV0}$ , allowing two pooling layers to complete within the same cycle without worsening the pipeline cycle, so we group POOL-0 and POOL-1 together. Similarly, we group CONV-4 and CONCAT layers together. Subsequently, CONV1-3 are three independent layers, thus grouped to compute simultaneously within one cycle. The group strategy results in 4 layer groups, offering a 50% reduction in the first-batch latency metric.

Algorithm 1 delineates the convolution operator-level HT dataflow scheduling algorithm. Since the hardware exhibits consistent behavior across different batches, the accelerator can reuse the same pseudo-instructions, so Algorithm 1 initially generates a saturated state pipeline throughout all layer groups, followed by the backend stage controlling the pipeline's filling and emptying processes. Lines 2-5 describe the scheduling strategy, where LayerGroupList is a list that arranges the layer groups obtained by the layer grouping strategy based on the topological order. In line 9, RepLayer.Run signifies the launch of a computational task, which, within PIMCOMP, is essentially a convolution operator, thus enabling fine-grained convolution operator-level scheduling to exploit hardware compute parallelism. The Run method includes reading one sliding window of the input feature map with consideration of data reuse, triggering the computation of each AG within the replica, and accumulating

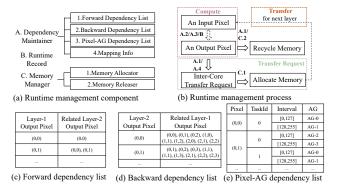


Fig. 9: Pixel-level runtime management strategy.

partial sums between AGs. If this layer is fused with an activation layer, the activation operation will be completed instantly using the VFUs. The result of this task, i.e., a new output pixel, will be stored back to global memory or retained on the chip based on the capacity of the on-chip local memory. Lines 12-14 indicates that layers beyond convolutional and fully-connected layers are also allocated to multiple cores to enhance computational parallelism. Line 17 entails the removal of layers that have completed computational tasks.

#### B. Low-Latency Mode

In the LL mode, to expedite the attainment of final results, each layer promptly transmits output results to subsequent layers via on-chip connections. This operational characteristic poses two challenges for compilation: I. On-chip communication overhead, and II. Runtime management of computational tasks and on-chip storage. We address the former by achieving a balance between computation and communication through layout-computation mapping, as detailed in Section VI. The latter challenge is unique to the fine-grained pipeline of PIM on-chip architecture. In this mode, aside from the first layer's input being read from global memory, the input data for other layers come from the outputs of their predecessor layers during runtime. Consequently, each AG stores the received data in local memory. These data cannot be immediately utilized but need to wait until enough data is received to perform the corresponding computational task. Due to the limited capacity of the on-chip local memory, the allocation of new memory block for received data cannot be indefinite; timely clearance of subsequently unused data is necessary to reclaim memory. Furthermore, in DNNs, a layer may have multiple predecessor and successor layers, each comprising multiple replicas and each replica composed of multiple AGs, thus compounding the complexity of this process.

To efficiently control the computation process, optimize memory usage, and reduce data transmission overhead, we propose the pixel-level runtime management strategy as illustrated in Fig. 9, which serves as a virtual runtime mechanism in software. Since each computational task outputs a complete pixel, we designate the pixel as the unit for memory allocation and data transmission. Fig. 9(a) illustrates the components required for management. With data transmission occurring between cores, data dependencies existing between replicas of

Algorithm 2: Low-latency dataflow scheduling.

```
Input: LayerQueue
1 TransReqQueue ← Queue()
2 while not LayerQueue.Empty() do
      Layer ← LayerQueue.Pop()
3
      foreach RepLayer in Layer.ReplicationList do
4
          if RepLayer.Ready(RepLayer.TaskIdx) then
5
              Result \leftarrow RepLayer.Compute(RepLayer.TaskIdx)
6
              TransReq \leftarrow RepLayer.TransmitRequest(Result)
              TransReqQueue.Append(TransReq)
              RepLayer.TaskIdx += 1
              Layer. TaskIdx += 1
10
      if Layer.TaskIdx < Layer.TaskNum then
11
          LayerQueue.Append(Layer)
12
      if Meet Transmit Conditions then
13
          PerformTransmit(TransReqQueue)
14
```

layers, and the granularity of transmission and allocation being the pixel, we establish multi-level dependencies through the dependency maintainer. The forward dependency list records the relationship between a layer's output pixels and its successor layers' output pixels. Inversely, the backward dependency list documents the relationship between a layer's output pixels and its predecessor layers' output pixels. The pixel-AG dependency list records the correspondence between AGs and input pixels across different computational tasks. The mapping info is the mapping relationship between AGs and cores obtained in Section VI. Additionally, we design the runtime record to log the address of each pixel within each core. To achieve flexible memory allocation, *PIMCOMP* adopts a heap-based memory manager for allocation and release.

Fig. 9(b) illustrates the management process. When a replica of a layer receives an input pixel, if *PIMCOMP* detects the feasibility of performing a computational task, it controls the AGs of this replica to calculate. For the result output pixel, we generate inter-core transmission requests based on A.1 and A.4, ensuring no redundant transmission between any two cores. Subsequently, these requests invoke C.1 for allocating local memory addresses for each pixel in the receiving core. Finally, data transmissions are carried out at an appropriate time, with these transmitted data serving as inputs for successor layers, thus re-entering the process. Upon completion of each computational task, we utilize A.1 and C.2 to delete pixels that will no longer be used by any other replicas, thus reclaiming memory.

Based on this foundation, we propose the LL dataflow scheduling algorithm, as shown in Algorithm 2. We also adopt convolution operator-level scheduling. The input parameter LayerQueue is a queue of layers arranged in the topological order. In lines 5 and 6, RepLayer.Ready and RepLayer.Compute correspond to the Compute process in Fig. 9(b). Unlike the RepLayer.Run method in the HT mode, the Compute method merely controls the computation process without engaging in data transmission. Regarding transmission requests, we do not execute them directly due to the reasons illustrated in Fig. 3, where frequent data transmissions can impact parallelism. Instead, we adopt a

TABLE V: Configuration of three existing architectures.

	#chip	#core	#crossbar	Crossbar array size	Cell precision	Capacity
Arch-A [5]	1	168	96	128×128	2-bit	63MB
Arch-B [22]	1	138	128	128×128	2-bit	69MB
Arch-C [33]	16	4	8	512×1024	2-bit	64MB

heuristic centralized communication mechanism: we cache data transmission requests first (lines 7 and 8) and initiate data transmission only when the number of transmission requests in the system reaches a certain threshold or when the last layer is traversed (lines 13 and 14). The request information for data transmission includes the data, the two cores involved in communication, and the pre-assigned receiving address. We ensure that memory is not reclaimed or overwritten before data transmission occurs.

#### VIII. EVALUATIONS

# A. Experimental Setup

To validate the generality of *PIMCOMP*, we instantiate the abstract hardware using three different architectures with the corresponding configurations presented in Table V and additional parameters referenced from existing literature [5]. We expand the number of chips of [33] to 16 to ensure sufficient PIM resources to accommodate complete DNN models and avoid weight rewriting. The power data for the crossbar arrays and VFU units come from [22]. The memory and router modules are simulated using CACTI [34] and Orion 3.0 [35] to obtain power data, which are then scaled to the 32nm technology node.

For the comparison of compilation methods, we select *SongC* [18], *PUMA* [22], and *Polyhedral* [16] as baselines, whose characteristics are elaborated in Section II. We implement these three methods faithfully within our compiler. Our comparison focuses on the impact of differences in scheduling granularity, resource allocation, data layout, and dataflow scheduling on performance. Therefore, the optimization methods designed by *PIMCOMP* for unfolding weight data, alleviating communication overhead, and reducing memory access are applied equally to each implementation. We conduct simulation through the profiler described in Section IV, which provides performance metrics such as inference latency, throughput, and energy consumption, along with detailed runtime information, including resource utilization, global memory access volume, and local memory overhead.

Four common networks are selected as benchmarks: vgg8 [36] and resnet18 [37] trained on the MNIST dataset, and resnet34 [37] and googlenet [38] trained on the ImageNet datasets. The weight data for these models are quantized to 16-bit fixed-point numbers. In the HT scenario, the batch size is set to 128, while in the LL mode, the batch size is 1.

#### B. End-to-End Inference Results

1) High Throughput Mode: Fig. 10 presents the throughput and resource utilization of different compilation strategies across three architectures in the HT scenario. *PIMCOMP* achieves significant enhancement across all benchmarks on all

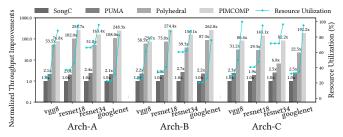


Fig. 10: Throughput and resource utilization of HT scenario.

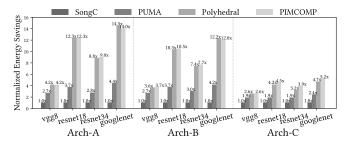


Fig. 11: Energy savings of HT scenario.

architectures. Both Polyhedral [16] and PIMCOMP employ weight replication methods and sample-granularity pipelining to increase hardware computation parallelism, resulting in throughput improvements of  $44.7\times$  and  $149.5\times$  over Song C [18], respectively. However, Polyhedral [16]'s mapping and scheduling granularity is at the layer level, where each core can only accommodate one replica of a layer, leading to idle resources. Regarding resource utilization, PIMCOMP defines the array group as the mapping granularity, enabling a finer layout mapping and allowing for a higher weight replication factor (with an average utilization improvement of 38.8% over *Polyhedral* [16]). In terms of scheduling granularity, PIMCOMP initiates a sliding window computation task each time, ensuring that arrays without structural conflicts within the core can operate in parallel. Consequently, PIMCOMP achieves an average throughput improvement of 3.3× over Polyhedral [16].

Fig. 11 compares the inference energy in the HT mode. PIMCOMP achieves significant energy savings across all three architectures, primarily because PIMCOMP boosts throughput, reducing total inference time and lowering the system's leakage energy. In the HT mode, the energy consumption of PIMCOMP and Polyhedral [16] are nearly identical because both significantly reduce the time to complete inference, making dynamic energy the dominant of total energy. Since the computational workload for inferring the same network remains constant across different compilation methods, PIMCOMP and Polyhedral [16] exhibit similar dynamic energy and total energy consumption. Overall, PIMCOMP achieves an average improvement of  $9.0\times$  and  $7.7\times$  over SongC [18] on Arch-A and Arch-B, respectively, but only a 3.9× improvement on Arch-C. This is because Arch-C has fewer cores than Arch-A and Arch-B, resulting in lower system leakage power, while the larger crossbar arrays in Arch-C incur higher dynamic power. Consequently, as *PIMCOMP* reduces energy consumption by lowering leakage energy, the

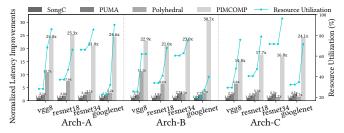


Fig. 12: Latency and resource utilization of LL scenario.

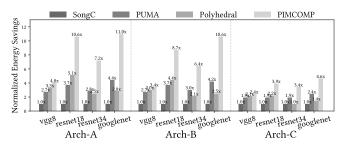


Fig. 13: Energy savings of LL scenario.

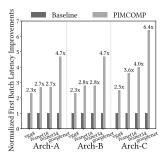
optimization effect on Arch-C is somewhat less pronounced than on Arch-A and Arch-B.

2) Low Latency Mode: Fig. 12 illustrates the inference latency and resource utilization of different compilation strategies in the LL scenario. *PIMCOMP* demonstrates a notable improvement over the other compilation methods on various architectures. Although *PUMA* [22] introduces inter-layer pipelining, it does not implement weight replication to utilize resources fully, resulting in imbalanced inter-layer execution times. The sample-granularity pipelining of *Polyhedral* [16] fails to meet the low-latency requirements. *PIMCOMP* designs inter-layer pipelines at the granularity of the convolution operator, achieving inference latency improvements of 21.8×, 9.8×, and 5.4× over *SongC* [18], *PUMA* [22], and *Polyhedral* [16], respectively, when the batch size is 1.

Fig. 13 displays the comparison of inference energy in the LL mode. *PIMCOMP* significantly reduces the latency, resulting in energy savings of  $5.6 \times$ ,  $2.0 \times$ , and  $2.3 \times$  compared with SongC [18], PUMA [22], and Polyhedral [16], respectively. The energy-saving effect in the LL mode is somewhat diminished compared with the HT mode, because the inference time per sample with a batch size of 1 is higher than the average time per sample with a batch size of 128, resulting in an increase in the inference leakage energy in the LL mode compared with the HT mode.

# C. System-Level Optimization

Figs. 14 to 17 showcase the effects of various individual system-level optimization techniques. Specifically, Fig. 14 illustrates the effect of the layer grouping strategy on first-batch latency in HT mode. Fig. 15 shows the impact of the flexible unfolding format, which leverages data reuse features, on global memory access in HT mode. Fig. 16 highlights the improvement in latency using the heuristic centralized communication mechanism of pixel-level runtime management strategy in the LL scenario. Fig. 17 presents the



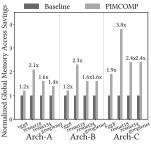
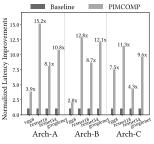


Fig. 14: First-batch latency Fig. 15: Global memory acord HT scenario. Fig. 15: Global memory acord HT scenario.



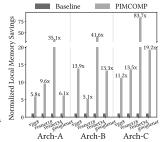


Fig. 16: Inference latency of Fig. 17: Local memory over-LL scenario. head of LL scenario.

optimization of local memory requirements within each core using the pixel-level runtime management strategy in the LL scenario. The baselines in Figs. 14 to 17 represent the performance metrics when *PIMCOMP* does not apply the respective optimization methods. In summary, *PIMCOMP* achieves end-to-end system-level optimization of DNN deployment tasks across dimensions of performance, resource, and memory.

# D. Compilation Time

Table VI shows the compilation time required by *PIMCOMP* on the Arch-A architecture. The population size in the genetic algorithm is 200, with up to 1000 iterations. The layout-computation mapping occupies most of the time, and the compilation process takes 19.2 minutes on average. Since compilation is a one-time effort and each model only requires compilation once, the overhead is acceptable.

# IX. CONCLUSION

We introduce *PIMCOMP*, an end-to-end DNN compiler designed for PIM accelerators. *PIMCOMP* demonstrates excellent versatility, adapting to hardware systems of various scales by abstracting the hardware architecture and the software-hardware interface. *PIMCOMP* integrates a multi-level optimizer for system-level performance and resource optimization to complete deployment efficiently. *PIMCOMP* bridges diverse DNN models and different PIM architectures, promoting the development of the PIM ecosystem.

#### REFERENCES

 S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE Annu. Int. Symp. Comput. Architect.*, p. 243–254, 2016.

TABLE VI: Compilation time (second).

		vgg8		resnet18		resnet34		googlenet	
		HT	LL	HT	LL	HT	LL	HT	LL
	$P^1$	0.0	0.1	0.1	0.1	0.6	0.6	0.6	0.6
	$M^2$	1034.4	414.8	1973.2	672.9	1542.1	725.5	2170.4	633.7
	$D^3$	3.0	0.5	2.7	0.4	9.5	16.4	7.6	14.7
	Total	1037.4	415.4	1976.0	673.4	1552.2	742.5	2178.6	649.0

<sup>&</sup>lt;sup>1</sup>P: layer partitioning. <sup>2</sup>M: layout-computation mapping. <sup>3</sup>D: dataflow scheduling

- [2] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Annu. IEEE/ACM Int. Symp. Microarchit.*, pp. 609– 622, 2014.
- [3] S.-L. Lu, T. Karnik, G. Srinivasa, K.-Y. Chao, D. Carmean, and J. Held, "Scaling the "memory wall"," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design*, p. 271–272, 2012.
- [4] X. Xu, J. Yu, T. Gong, J. Yang, J. Yin, D. Nian Dong, Q. Luo, J. Liu, Z. Yu, Q. Liu, H. Lv, and M. Liu, "First demonstration of oxrram integration on 14nm finfet platform and scaling potential analysis towards sub-10nm node," in *Annu. IEEE Int. Electron Devices Meeting*, pp. 24.3.1–24.3.4, 2020.
- [5] 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 *Proc. ACM/IEEE Annu. Int. Symp. Comput. Architect.*, p. 14–26, 2016.
- [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," in *Proc. ACM/IEEE Annu. Int. Symp. Comput. Architect.*, p. 27–39, 2016.
- [7] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined rerambased accelerator for deep learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, pp. 541–552, 2017.
- [8] T. Soliman, R. Olivo, T. Kirchner, C. D. I. Parra, M. Lederer, T. Kämpfe, A. Guntoro, and N. Wehn, "Efficient fefet crossbar accelerator for binary neural networks," in *IEEE Int. Conf. Appl. Sys. Archit. Process. (ASAP)*, pp. 109–112, 2020.
- [9] S. Jung, H. Lee, S. Myung, H. Kim, S. K. Yoon, S.-W. Kwon, Y. Ju, M. Kim, W. Yi, S. Han, B. Kwon, B. Seo, K. Lee, G.-H. Koh, K. Lee, Y. Song, C. Choi, D. Ham, and S. J. Kim, "A crossbar array of magnetoresistive memory devices for in-memory computing," *Nature*, vol. 601, pp. 211–216, Jan 2022.
- [10] A. Chen, S. Ambrogio, P. Narayanan, H. Tsai, C. Mackin, K. Spoon, A. Friz, A. Fasoli, and G. W. Burr, "Enabling high-performance dnn inference accelerators using non-volatile analog memory," in *IEEE Electron Devices Technol. Manuf. Conf. (EDTM)*, pp. 1–4, 2020.
- [11] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature Nanotechnology*, vol. 15, pp. 529–544, Jul 2020.
- [12] X. Sun, X. Wang, W. Li, L. Wang, Y. Han, and X. Chen, "Pimcomp: A universal compilation framework for crossbar-based pim dnn accelerators," in *Proc. Design Autom. Conf.*, pp. 1–6, 2023.
- [13] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated End-to-End optimizing compiler for deep learning," in USENIX Symp. Oper. Sys. Design Impl., pp. 578–594, 2018.
- [14] Y. Xiao, G. Ma, N. K. Ahmed, M. Capotă, T. L. Willke, S. Nazarian, and P. Bogdan, "End-to-end programmable computing systems," *Communications Engineering*, vol. 2, p. 84, Nov 2023.
- [15] Y. Xiao, S. Nazarian, and P. Bogdan, "Plasticity-on-chip design: Exploiting self-similarity for data communications," *IEEE Transactions on Computers*, vol. 70, no. 6, pp. 950–962, 2021.
- [16] J. Han, X. Fei, Z. Li, and Y. Zhang, "Polyhedral-based compilation framework for in-memory neural network accelerators," *J. Emerg. Tech*nol. Comput. Syst., vol. 18, sep 2021.
- [17] J. Ambrosi, A. Ankit, R. Antunes, S. R. Chalamalasetti, S. Chatterjee, I. E. Hajj, G. Fachini, P. Faraboschi, M. Foltin, S. Huang, W.-M. Hwu, G. Knuppe, S. V. Lakshminarasimha, D. Milojicic, M. Parthasarathy, F. Ribeiro, L. Rosa, K. Roy, P. Silveira, and J. P. Strachan, "Hardware-software co-design for an analog-digital accelerator for machine learning," in *IEEE Int. Conf. Reb. Comput. (ICRC)*, pp. 1–13, 2018.
- [18] J. Lin, H. Qu, S. Ma, X. Ji, H. Li, X. Li, C. Song, and W. Zhang, "Songe: A compiler for hybrid near-memory and in-memory many-core architecture," *IEEE Trans. Comput.*, pp. 1–14, 2023.

- [19] A. Drebes, L. Chelini, O. Zinenko, A. Cohen, H. Corporaal, T. Grosser, K. Vadivel, and N. Vasilache, "Tc-cim: Empowering tensor comprehensions for computing-in-memory," in *Int. Workshop on Poly. Compil.* Tech., 2020.
- [20] K. Vadivel, L. Chelini, A. BanaGozar, G. Singh, S. Corda, R. Jordans, and H. Corporaal, "Tdo-cim: Transparent detection and offloading for computation in-memory," in *Proc. Design Autom. Test Europe Conf. Exhibit.*, pp. 1602–1605, 2020.
- [21] A. Siemieniuk, L. Chelini, A. A. Khan, J. Castrillon, A. Drebes, H. Corporaal, T. Grosser, and M. Kong, "Occ: An automated end-toend machine learning optimizing compiler for computing-in-memory," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 41, no. 6, pp. 1674–1686, 2022.
- [22] 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, and D. S. Milojicic, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, p. 715–731, 2019.
- [23] A. A. Khan, H. Farzaneh, K. F. Friebel, L. Chelini, and J. Castrillon, "Cinm (cinnamon): A compilation infrastructure for heterogeneous compute in-memory and compute near-memory paradigms," arXiv:2301.07486, 2022.
- [24] W. Li, Y. Han, and X. Chen, "Mathematical framework for optimizing crossbar allocation for reram-based cnn accelerators," ACM Trans. Des. Autom. Electron. Syst., vol. 29, dec 2023.
- [25] S. Yoo and A. Jerraya, "Introduction to hardware abstraction layers for soc," in *Proc. Design Autom. Test Europe Conf. Exhibit.*, pp. 336–337, 2003.
- [26] X. Qiao, X. Cao, H. Yang, L. Song, and H. Li, "Atomlayer: a universal reram-based cnn accelerator with atomic layer computation," in *Proc. Design Autom. Conf.*, 2018.
- [27] Z. Zhu, H. Sun, T. Xie, Y. Zhu, G. Dai, L. Xia, D. Niu, X. Chen, X. S. Hu, Y. Cao, Y. Xie, H. Yang, and Y. Wang, "Mnsim 2.0: A behavior-level modeling tool for processing-in-memory architectures," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 42, no. 11, pp. 4112–4125, 2023.
- [28] J. Peng, H. Liu, Z. Zhao, Z. Li, S. Liu, and Q. Li, "Cmq: Crossbar-aware neural network mixed-precision quantization via differentiable architecture search," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 4124–4133, 2022.
- [29] Z. Zhu, J. Lin, M. Cheng, L. Xia, H. Sun, X. Chen, Y. Wang, and H. Yang, "Mixed size crossbar based rram cnn accelerator with overlapped mapping method," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, p. 1–8, 2018.
- [30] X. Peng, R. Liu, and S. Yu, "Optimizing weight mapping and data flow for convolutional neural networks on rram based processing-in-memory architecture," in *Proc. IEEE Int. Symp. Circuits Syst.*, pp. 1–5, 2019.
- [31] L. Han, P. Huang, Z. Zhou, Y. Chen, X. Liu, and J. Kang, "A convolution neural network accelerator design with weight mapping and pipeline optimization," in *Proc. Design Autom. Conf.*, pp. 1–6, 2023.
- [32] Y. Wang and X. Fong, "Benchmarking dnn mapping methods for the in-memory computing accelerators," *IEEE J. Emerg. Sel. Topics Circuits* Sys., vol. 13, no. 4, pp. 1040–1051, 2023.
- [33] W.-H. Huang, T.-H. Wen, J.-M. Hung, W.-S. Khwa, Y.-C. Lo, C.-J. Jhang, H.-H. Hsu, Y.-H. Chin, Y.-C. Chen, C.-C. Lo, R.-S. Liu, K.-T. Tang, C.-C. Hsieh, Y.-D. Chih, T.-Y. Chang, and M.-F. Chang, "A nonvolatile al-edge processor with 4mb slc-mlc hybrid-mode reram compute-in-memory macro and 51.4-251tops/w," in *IEEE Int. Solid-State Circuits Conf.*, pp. 15–17, 2023.
- [34] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," ACM Trans. Archit. Code Optim., vol. 14, jun 2017.
- [35] A. B. Kahng, B. Lin, and S. Nath, "Orion3.0: A comprehensive noc router estimation tool," *IEEE Embedded Systems Letters*, vol. 7, no. 2, pp. 41–45, 2015.
- [36] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," CoRR, vol. abs/1409.1556, 2014.
- [37] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vision Pattern Recog.* (CVPR), June 2016.
- [38] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vision Pattern Recog.* (CVPR), pp. 1–9, 2014.



Xiaotian Sun received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China. in 2021. He is currently pursuing the Ph.D. degree with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing. He is currently interested in accelerating deep neural networks on emerging storage devices.



**Xinyu Wang** received the B.E. degree from the School of Computer Science and Technology, Shandong University, Jinan, China, in 2022. He is currently pursuing the M.S. degree with the University of Chinese Academy of Sciences, Beijing, China. His research interests include computer architecture, processing-in-memory, and deep learning.



Wanqian Li received the B.S. degree in Microelectronics Science and Engineering from Nankai University, Tianjin, China. in 2020. She is currently pursuing the Ph.D. degree with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing. She is currently interested in automatic synthesis of processing-in-memory architectures and acceleration of large language model inference.



Yinhe Han (Member, IEEE) received the M.S. and Ph.D. degrees in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), in 2003 and 2006, respectively. He is currently a Professor with ICT, CAS. His main research interests are microprocessor design, integrated circuit design, and computer architecture.



Xiaoming Chen (Member, IEEE) received the B.S. and Ph.D. degrees in electronic engineering from Tsinghua University, Beijing, China, in 2009 and 2014, respectively. He is currently a Professor with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His current research interest is focused on design automation for integrated circuits and PIM architectures.