SIFMAP: Parallel CNN Inference using Sub-Input-Feature Maps in PIM

Tong Wu, Shuibing He, Weijian Chen, Xuechen Zhang, Tuo Shi

***Abstract*—Processing-in-memory (PIM) accelerators perform in-situ dot-product operations between input feature maps (IFMs) and convolutional neural network (CNN) weights to accelerate inference. The efficiency of these computations is heavily influenced by the weight mapping strategy. However, existing approaches often adopt IFM-oriented mapping schemes that provide limited parallelism, leading to high latency and low energy efficiency. To address this issue, we propose SIFMAP, a subIFM-oriented weight mapping framework designed to enable highly parallel CNN inference. SIFMAP leverages the observation that different regions within an IFM contribute unevenly to computation. It first partitions the IFM into fine-grained sub-maps, or subIFMs, based on their computational characteristics. Then, it applies a reinforcement learning (RL) technique to determine optimal weight mapping strategies tailored to each subIFM, optimizing global performance. Furthermore, SIFMAP introduces a subIFM-oriented resource allocation scheme to further boost computational parallelism across subIFMs. Experimental results show that SIFMAP reduces inference latency by $4.8\times$ and improves energy efficiency by $5.2\times$ compared to state-of-the-art mapping methods under equivalent resource constraints.**

***Index Terms*—Pattern-aware, Shift and duplicate kernel, Reinforcement Learning, Processing-in-memory**

I. INTRODUCTION

As a classic model, the convolutional neural network (CNN) continues to be extensively employed across computer vision fields, including image recognition [1], object detection [52], and even advanced large model scenarios [38]. As CNN models grow in scale, their computing and memory demands increase, posing significant challenges for traditional von Neumann architectures due to excessive data movement between processor and memory [45], [56]. To address these issues, processing-in-memory (PIM) technologies, such as resistive random access memory (ReRAM), have been developed [40], [6], [44].

Convolution operations essentially involve numerous dot-product operations between input feature maps (IFMs) and CNN weights. ReRAM-based accelerators are used to execute these dot-product operations in the analog domain using memristor-based crossbars [26], [59], [12], thereby accelerating CNN inference. The efficiency of this process heavily depends on the mapping scheme of weights onto the crossbar.

T. Wu, S. He, and W. Chen are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: wu.tong@zju.edu.cn; heshuibing@zju.edu.cn; weijianchen@zju.edu.cn).

X. Zhang is with the Department of Information Technology, Kennesaw State University, Kennesaw, GA 30144, USA (e-mail: xzhang54@kennesaw.edu).

T. Shi is with Zhejiang Laboratory, Hangzhou 311122, China (e-mail: shituo@zhejianglab.org).

Shuibing He is the corresponding author.

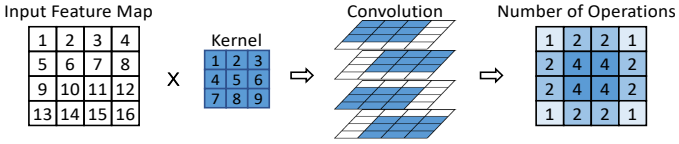


Fig. 1. An example of an IFM computation pattern, i.e., the number of operations on each IFM element.

Existing mapping schemes are generally suboptimal due to their low input parallelism, resulting in high latency and low energy efficiency. Specifically, the widely used image to column (Im2col) method maps all CNN kernels onto crossbars once and performs dot products through multiple input cycles [40], [44]. For example, a 3×3 convolution kernel performs convolution operations with padding of 1 and stride of 1 on a 32×32 IFM, requiring a total of 1024 input cycles [17]. To reduce input cycles and boost input parallelism, shift and duplicate kernel (SDK) methods replicate kernels and leverage shift mapping, enabling less input cycles at the cost of increased resource consumption [62], [35]. However, they degrade to Im2col when the crossbar space is too constrained to have a desired number of replicas, as shown in (§II-B).

We notice that, during convolution, different elements within an IFM contribute to varying numbers of operations. For example, as shown in Figure 1, each element in a 4×4 IFM participates in a different number of operations (each operation is the multiplication between the convolution kernel and the kernel overlay on IFM). Here, we name this "map of number of operations" in Figure 1 as an *IFM computation pattern*. Inspired by this, we can partition the original IFM into finer-grained sub-maps, which we define as **subIFM**. Compared to the original IFM, subIFMs are smaller and can be parallelized with each other, while maintaining only essential numbers of operations. Therefore, utilizing subIFMs presents notable advantages for exploring highly parallel CNN inference.

In this paper, we propose SIFMAP, a subIFM-oriented weight mapping framework, which partitions each IFM into subIFMs considering patterns and designs weight mapping to enable subIFM parallelism. The core idea is to divide inputs with similar operation numbers into the same subIFM and employ uniform weight replica strategy while different strategies for distinct subIFMs, thereby maximizing input parallelism and realizing dynamic weight mapping. However, building such an efficient mapping scheme involves three significant challenges.

First, **the patterns in IFMs can vary among different CNN layers and models.** It poses a challenge in identifying subIFMs, considering that patterns are influenced by various

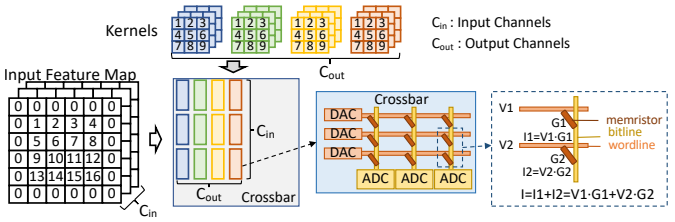


Fig. 2. Weight mapping scheme for ReRAM crossbar circuits.

factors such as IFM sizes, kernel sizes, padding sizes, etc. Therefore, it is essential to formulate a model to identify the pattern and direct the partition of an IFM to subIFMs.

Second, **mapping subIFMs to crossbars using the existing greedy algorithms cannot achieve global optimization.** Since subIFMs introduce a more complex and expansive decision space, selecting the optimal weight mapping strategy for all subIFMs becomes a challenge. Directly applying existing greedy-based methods is suboptimal and low-efficient because these methods only focus on individual CNN layer while ignoring the overall system performance [62], [35]. This necessitates a more automated and efficient method to select the optimal mapping strategy for all subIFMs, ensuring global performance optimization.

Third, **the existing crossbar allocation schemes do not exploit subIFM-level parallelism.** Current ReRAM-based accelerators employ an IFM-oriented crossbar allocation scheme, which arranges the weights for each IFM column by column on the crossbar. This method may map weights for different subIFMs onto the same crossbar, serializing their computation and hindering input parallelism.

To address the above challenges, SIFMAP employs a three-stage process. First, it establishes a general model for IFMs and partitions each IFM into three types of subIFMs—corner, edge, and center—based on their patterns. Second, SIFMAP employs an automated reinforcement learning (RL) algorithm to select proper mapping strategies for specific subIFMs, achieving global optimal performance. Third, SIFMAP adopts a subIFM-oriented crossbar allocation scheme to ensure computational parallelism among subIFMs, thereby further improving performance.

In summary, this paper makes the following contributions:

- We introduce SIFMAP, the first subIFM-oriented mapping framework that leverages the IFM computation pattern to enhance input parallelism for CNN inference.
- We introduce an RL algorithm to automatically select the optimal mapping strategy for different subIFMs, achieving global optimal performance and energy efficiency.
- We develop a subIFM-oriented crossbar allocation scheme that further enhances computational parallelism.
- We evaluate SIFMAP with five models on three datasets. Experimental results show that SIFMAP reduces latency by $4.8\times$ and improves energy efficiency by $5.2\times$ on average, compared to state-of-the-art mapping schemes under equivalent resource constraints.

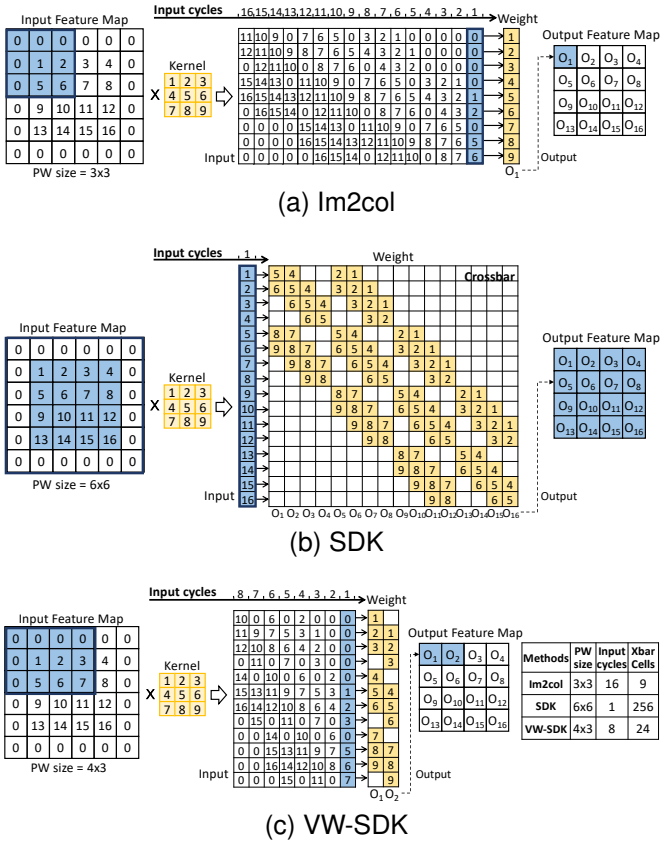


Fig. 3. Mapping convolution kernels onto a ReRAM crossbar with existing mapping schemes. (a) Im2col. (b) SDK. (c) VW-SDK. The IFM is 4×4 in size, with zero-padding on all sides. The kernel size is 3×3 and the stride length is one.

II. BACKGROUND

A. ReRAM-based Accelerators for CNN Inference

Convolutional neural networks (CNNs) demonstrate substantial performance in traditional classification [1], [15], detection [27], [52], and extraction tasks [16], [25], while also contributing to advanced large model scenarios. For instance, 2D convolution operations constitute over 80% of the total operations in Stable Diffusion, making them the primary model inference bottleneck [38], [13], [29]. As the scale of CNNs continues to grow, numerous convolution operations, namely dot-product operations between input feature maps (IFMs) and convolution kernels, cause severe computational and memory bottlenecks in traditional computer architectures [45], [56]. As practical processing-in-memory devices, ReRAM-based accelerators can perform these dot products with low-energy, low-latency in-situ computation, providing a viable solution for accelerating CNN inference [26], [40], [2], [58].

Figure 2 shows a basic ReRAM crossbar circuit with wordlines, bitlines, memristors, digital-to-analog converters (DACs), and analog-to-digital converters (ADCs). During convolutions, kernel weights are pre-stored as conductance values G in memristors, while IFMs are converted to voltage values V on wordlines via DACs. Weights of different input channels are mapped to the same crossbar column, and those of different output channels to adjacent columns [35]. Each memristor produces a current I according to Ohm's law ($I = V \times G$).

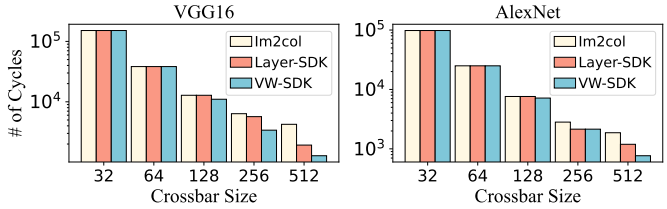


Fig. 4. The input cycles of existing mapping methods on VGG16 and AlexNet.

The currents from the same bitline are summed and converted to digital values by ADCs, marking the completion of dot products. Since crossbar size is limited while CNN weights are numerous, convolution operations often require multiple crossbars and peripheral circuits to work together over several input cycles. Thus, the mapping scheme of weights onto crossbars significantly impacts CNN inference efficiency.

B. Existing Weight Mapping Methods

Image to column (Im2col). Im2col is a straightforward mapping scheme that maps all CNN kernels onto crossbars once and performs dot products through multiple input cycles [40], [44]. For a $K \times K$ kernel, Im2col flattens the kernel into a column vector of size $K^2 \times 1$, which is then mapped to a bitline of a crossbar, as shown in Figure 3a. This neat arrangement results in the lowest crossbar consumption (i.e., 9×1). However, it suffers from higher latency and lower energy efficiency due to the need for 16 input cycles.

Shift and duplicate kernel (SDK). To reduce input cycles, SDK duplicates several CNN kernels and maps them to adjacent crossbar columns in a shifting manner, enabling all dot products to be calculated in a single cycle [62], as Figure 3b illustrates. This approach reduces latency and improves energy efficiency at the cost of significantly increased crossbar consumption used for kernel duplication (i.e., 16×16). Since mapping all CNN layers with SDK consumes excessive resources, Layer-SDK [62] proposes to selectively deploy the SDK only on a few convolutional layers in resource-constrained scenarios. Based on this, the state-of-the-art VW-SDK [35] employs a greedy algorithm to select appropriate SDK strategies for each convolutional layer, as illustrated in Figure 3c. It performs the same convolution operation within 8 input cycles with a 12×2 crossbar consumption.

For convenience of comparison, we adopt the concept of **parallel-window (PW)** from SDK [62] to evaluate the input parallelism of different mapping methods. A PW denotes a group of windows that share portions of the convolution kernels on the input feature maps (IFMs) and perform dot products in a single cycle. A larger PW size indicates higher input parallelism, resulting in fewer input cycles, while smaller PW sizes require more cycles to complete the computation. The PW size ranges between the kernel size and the full IFM size.

- **Kernel-size PW**: Corresponds to a simplified mapping scheme equivalent to Im2col (Figure 3a).
- **IFM-size PW**: Enables all inputs to be processed in a single cycle, effectively achieving the full parallelism of SDK (Figure 3b).

TABLE I
THE PW SIZES OF EXISTING METHODS ON VGG16. $C1$ – $C13$ DENOTE CONVOLUTIONAL LAYERS. RATIO REPRESENTS THE PROPORTION OF PW SIZE EQUAL TO KERNEL SIZE.

Methods	XB Size	C1	C2	C3	C4	C5-7	C8–C13	Ratio
Im2col	—	3×3	3×3	3×3	3×3	3×3	3×3	100%
Layer-SDK	128×128	3×3	3×3	3×3	3×3	3×3	3×3	100%
	256×256	4×4	3×3	3×3	3×3	3×3	3×3	92.3%
	512×512	4×4	4×4	4×4	3×3	3×3	3×3	76.9%
VW-SDK	128×128	4×3	4×3	3×3	3×3	3×3	3×3	84.6%
	256×256	4×4	4×4	4×3	4×3	3×3	3×3	69.2%
	512×512	3×10	4×4	4×4	4×4	4×3	3×3	46.2%

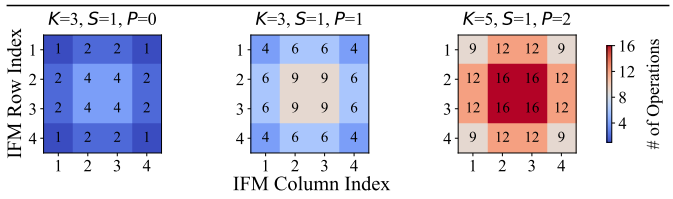


Fig. 5. Heatmaps presenting the number of operations for each element in the 4×4 IFM under varying convolutional parameters.

- **In-between-size PW**: Offers opportunities to optimize system latency and energy efficiency under specified resource constraints (Figure 3c).

Table I summarizes the mapping strategies (i.e., PW selections) produced by existing methods for thirteen 3×3 convolutional layers (C1–C13) in VGG16 [17]. The results reveal that when the crossbar size is constrained, SDK-based methods often degrade to an Im2col-like mapping, thereby forfeiting the latency-reduction advantages of SDK. Specifically, 46.2% to 100% of the layers adopt a 3×3 PW size–identical to the kernel size–with this trend worsening as the crossbar size decreases. As a result, the number of input cycles becomes equivalent to that of Im2col, eliminating any acceleration benefit, as illustrated in Figure 4. This degradation occurs because existing approaches restrict PW selection to fit within a single crossbar, overlooking the potential of collaborative mapping across multiple crossbars. Consequently, weight mapping remains a critical bottleneck that limits the achievable parallelism of CNN inference.

III. Motivation and Challenges

A. IFM Computation Pattern

Since each convolution kernel processes the input feature map (IFM) in a sliding manner, and the kernel overlay regions overlap with each other, the number of dot-product operations (denoted as N_{ops}) varies across positions on the IFM. Figure 1 illustrates the N_{ops} values for different IFM elements (i.e., 1 to 16). Assuming the IFM size is 4×4, the kernel size K is 3×3, the padding size P is zero, and stride length S is one, we observe that the N_{ops} for each position on the IFM ranges from 1 to 4. This observation is prevalent across most convolutional scenarios, except when $K = 1 \times 1$. Figure 5 illustrates the N_{ops} distribution via heatmaps for a 4×4 IFM under varying convolutional parameters (i.e., K, S, and P), demonstrating that N_{ops} changes depending on the position.

We refer to this spatially uneven operation discrepancy across different IFM elements as the *IFM computation pattern*. Owing to this inherent computation pattern, corner, edge, and

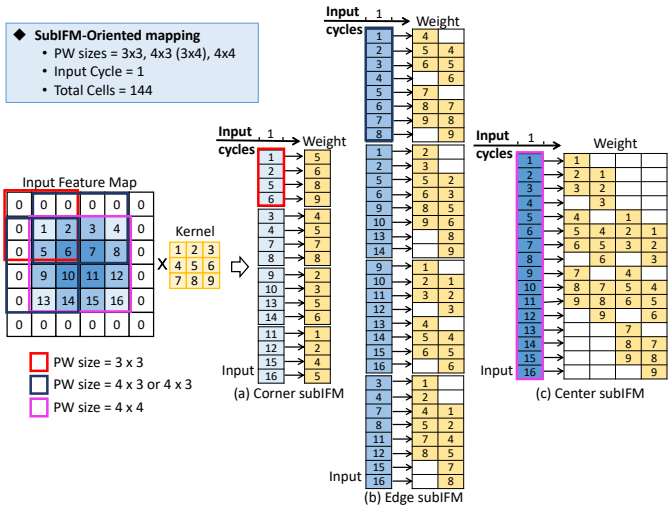


Fig. 6. The subIFM-oriented weight mapping.

center regions incur vastly different convolution workloads. If weight mapping is performed on the full IFM, all weights must be mapped in accordance with the maximum N_{ops}, which introduces substantial redundant resource overhead. This problem motivates us to partition the original IFM into fine-grained independent regions termed subIFMs. These subIFMs are smaller, support parallel processing, and retain only the necessary operations matching their spatial positions, thereby eliminating resource waste. This fine-grained partitioning delivers higher computational efficiency and better parallelizability.

Figure 6 illustrates a naive subIFM-oriented weight mapping scheme, which partitions the original IFM into nine subIFMs according to their N_{ops} values. We assign PW sizes of 3×3 to corner subIFMs, 4×3 and 3×4 to edge subIFMs, and 4×4 to the center subIFM. Note that each IFM contains horizontal and vertical edges, leading the edge subIFMs to adopt two PW sizes. Since these two edge PW sizes yield identical computational performance, we use a unified PW size representation for all edge subIFMs in the subsequent analysis for simplicity. With this configuration, the convolution completes in a single input cycle–matching the efficiency of the SDK case in Figure 3b–yet consumes only 56% of the crossbar resources required by SDK. Moreover, because each subIFM is smaller than the full IFM, using a given PW on these smaller regions requires fewer input cycles than applying the same PW across the entire IFM.

B. Challenges

An effective subIFM-oriented weight mapping framework must meet three requirements. First, it needs to identify IFM patterns and divide an IFM into subIFMs appropriately. Second, it employs an automated method to determine the optimal mapping strategy (i.e., PW sizes) for all subIFMs based on the global system feedback. Third, it allows multiple crossbars to collaboratively map weights and ensure computing parallelism. Building such a framework introduces three key challenges:

Challenge 1: The IFM computation pattern can vary across CNN layers or models. Factors influencing the number of dot-product operations (i.e., N_{ops}) difference include IFM sizes, kernel sizes, convolution strides, and padding sizes, all of which vary significantly, making it challenging to partition the IFM into appropriate subIFMs based on N_{ops}. This challenge motivates us to build a model that accounts for these factors to serve as a foundation for subIFM-oriented PW selection.

Challenge 2: Directly applying existing PW selection methods in subIFM-oriented situations is suboptimal and low-efficient. Existing SDK approaches adopt greedy-based methods to select PW sizes layer by layer [62], [35], which ignore the global hardware feedback. Tuning PW sizes affects both intra-layer computation latency and inter-layer communication overhead such as data transmission and buffer read/write latency. Existing methods only focus on intra-layer computation latency and neglect the impact of inter-layer communication on overall performance in real hardware deployments, thus easily yielding suboptimal solutions. Moreover, compared to existing IFM-oriented methods, subIFMs introduce a significantly larger search space. For an N-layer CNN model with C candidate PW sizes, assuming each layer's IFMs can be divided into M subIFMs, the PW selection search space becomes C^{N+M}. This space grows exponentially for complex models and large IFMs. For instance, EfficientNet [18] contains 49 convolutional layers with kernel sizes greater than 1, and the ImageNet dataset [8] provides 224×224 IFMs. Existing strategies suffer from excessive search overhead when applied to large-scale models, as verified in our experiments (§V-D). This challenge motivates the adoption of an automated method to efficiently determine the mapping strategy, focusing on achieving global optimality.

Challenge 3: The existing crossbar allocation scheme hinders input parallelism considering our subIFM-oriented weight mapping. Existing accelerators typically adopt a layer-oriented crossbar allocation scheme, mapping each CNN layer's weights to adjacent columns of one or more crossbars to maximize utilization [64], [41]. With subIFM-oriented weight mapping, weights for different subIFMs within the same IFM may share a crossbar. For example, mapping the weights in Figure 6 onto the same crossbar column by column increases the input cycles from one to nine, as the nine subIFMs have different input data. This mapping reduces input parallelism, as different inputs cannot be processed in parallel on the same crossbar wordline. Thus, a new subIFM-oriented crossbar allocation scheme to enable parallel computation is essential.

IV. Design

A. System Overview

We propose SIFMap, a subIFM-oriented weight mapping framework, which enables subIFM parallelism for high-efficient CNN inference. SIFMap is designed based on three key principles: (1) **SubIFM Identification**: Exploit IFM computation patterns to partition input feature maps into subIFMs, enabling fine-grained weight mapping and improved input parallelism. (2) **Optimized PW Selection**: Design an automated subIFM-oriented PW selection method to ensure globally optimal performance. (3) **Parallel Crossbar Allocation**: Allocate crossbars based on subIFMs to fully exploit subIFM-level parallelism. Figure 7 illustrates the overview of SIFMap, comprising a three-stage optimization process:

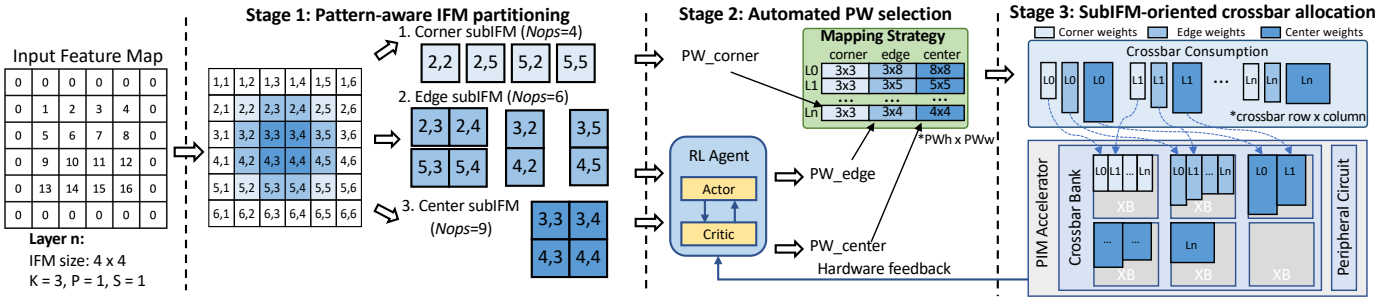


Fig. 7. The overview of SIFMAP mapping framework.

Stage 1: Pattern-aware IFM partitioning: Guided by principle (1), SIFMAP partitions IFMs into three types of subIFMs—corner, edge, and center—to facilitate subIFM-oriented weight mapping. A mathematical model is used to compute the map of the number of operations (i.e., N_{ops}) guiding the partition process to obtain subIFMs (§IV-B).

Stage 2: Automated PW selection: Following principle (2), SIFMAP employs a reinforcement learning (RL) method to globally improve input parallelism within specified resource constraints (§IV-C). For small corner subIFMs, SIFMAP uses the kernel size as PW sizes directly. For large edge and center subIFMs, the RL agent selects optimal PW sizes based on the overall system feedback. This stage generates a mapping strategy with appropriate PW sizes for each subIFM across all convolutional layers.

Stage 3: SubIFM-oriented crossbar allocation: Following principle (3), SIFMAP adopts a subIFM-oriented crossbar allocation scheme to allocate weights for different subIFMs of the same IFM to separate crossbars, thus ensuring subIFM-level input parallelism (§IV-D).

Additionally, we propose a SIFMAP compiler framework to facilitate the implementation of SIFMAP in real-world systems (§IV-E).

B. Pattern-aware IFM Partitioning

1) IFM analysis and modeling: As the foundation for subIFM-oriented weight mapping, it is essential to partition the IFMs into finer-grained subIFMs based on observed computation workload differences (§III-A). Thus, we first formulate a mathematical model to accurately estimate the number of operations (N_{ops}) of each IFM element by considering all related convolutional factors.

Figure 8 provides an example where the IFM size is $W \times H$, the convolution kernel size is $K \times K$, the padding size is P, and the stride length is S. To describe this process, we denote the padded IFM as IFM'. The upper left corner of IFM' is $(1,1)$, with the i-axis extending to the right and the j-axis extending downward (i.e., the black axis). Each position on IFM' is identified by a two-dimensional coordinate (i,j). The operation count at (i,j), denoted as $N_{ops}(i,j)$, can be derived using the known parameters. Since IFM' has a symmetrical layout, we only analyze its upper left quadrant while results for other regions are mirrored. We consider three scenarios:

When both i and j are smaller than K: $N_{ops}(i,j)$ is calculated using Equation 1.

$$N_{ops}(i,j) = \left\lceil \frac{i}{S} \right\rceil \times \left\lceil \frac{j}{S} \right\rceil \quad , \quad 0 < i < K, 0 < j < K \quad (1)$$

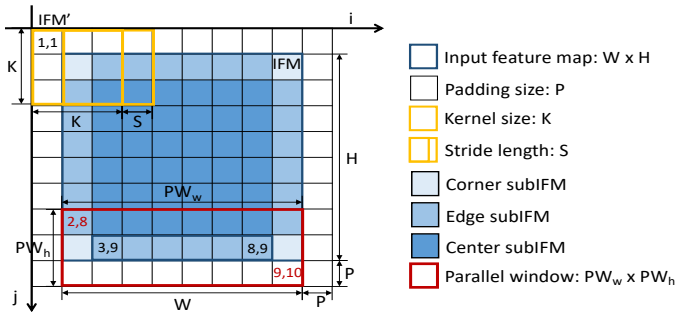


Fig. 8. The IFM partitioning and initial PW value assignment.

When one of i or j lies between K and half the size of IFM': $N_{ops}(i,j)$ is calculated using Equation 2.

$$\begin{cases} N_{ops}(i,j) = \left\lceil \frac{K}{S} \right\rceil \times \left\lceil \frac{j}{S} \right\rceil, & K \le i \le \left\lceil \frac{W}{2} + P \right\rceil, 0 < j < K \\ N_{ops}(i,j) = \left\lceil \frac{i}{S} \right\rceil \times \left\lceil \frac{K}{S} \right\rceil, & 0 < i < K, K \le j \le \left\lceil \frac{H}{2} + P \right\rceil \end{cases} \quad (2)$$

When both i and j are between K and half the size of IFM': $N_{ops}(i,j)$ is calculated using Equation 3.

$$N_{ops}(i,j) = \left\lceil \frac{K}{S} \right\rceil \times \left\lceil \frac{K}{S} \right\rceil, K \le i \le \left\lceil \frac{W}{2} + P \right\rceil, K \le j \le \left\lceil \frac{H}{2} + P \right\rceil \quad (3)$$

Finally, we integrate the above equations into Equation 4, which provides a general formula for $N_{ops}(i,j)$.

$$N_{ops}(i,j) = \left\lceil \frac{\min(i,K)}{S} \right\rceil \times \left\lceil \frac{\min(j,K)}{S} \right\rceil, i \le \left\lceil \frac{W}{2} + P \right\rceil, j \le \left\lceil \frac{H}{2} + P \right\rceil \quad (4)$$

This formula incorporates position coordinates and constant features of IFMs and kernels, enabling efficient estimation of operation counts at each IFM position. In the example shown in Figure 7, elements $(2,2)$, $(2,3)$, and $(3,3)$ have N_{ops} values of 4, 6, and 9, respectively. Note that convolutions typically involve multiple input and output channels. Elements at the same position within each input channel of the IFM share the same N_{ops}, and output channels depend only on the number of kernels; neither affects N_{ops} computation nor subIFM partitioning. Thus, channels are not considered in the above equations. When performing practical weight mapping, our method complies with classic channel placement strategies. Specifically, weights of different input channels are arranged column-wise, while weights corresponding to different output channels occupy adjacent columns [35], as depicted in Figure 2.

Due to the nature of convolution operations, the N_{ops} values across the IFM follow a centrosymmetric distribution. Specifically, the N_{ops} values are highest in the center region, moderate at the edges, and lowest at the corners. The pattern is illustrated in Figure 1. Leveraging this insight, we can

implement on-demand weight mapping tailored to different IFM regions to meet their specific computational requirements. Consequently, we partition each IFM into three types of subIFMs: center, edge, and corner, as Figure 7 illustrates.

2) Initial PW Selection: For each subIFM, the initial PW size is selected via the SDK method [62] to ensure all computations finish within a single input cycle. This value serves as the starting point for the subsequent automated PW selection process. We represent a given subIFM with the coordinates using its top-left element (i_1, j_1) and bottom-right element (i_2, j_2). Then, its initial PW size can be calculated using Equation 5:

$$\begin{cases} x_1 = \max\left(1-P, i_1 - \frac{K-1}{2}\right), y_1 = \max\left(1-P, j_1 - \frac{K-1}{2}\right) \\ x_2 = \min\left(W+2P, i_2 + \frac{K-1}{2}\right), y_2 = \min\left(H+2P, j_2 + \frac{K-1}{2}\right) \end{cases} \tag{5}$$

where (x_1, y_1) and (x_2, y_2) represent the top-left and bottom-right elements of the PW. The width PW_w and height PW_h of the PW can be determined using Equation 6.

$$\begin{cases} PW_w = x_2 - x_1 + 1 \\ PW_h = y_2 - y_1 + 1 \end{cases} \tag{6}$$

Figure 8 shows an example of an edge subIFM starting from $(3, 9)$ and ending at $(8, 9)$. According to Equations 5 and 6, the top-left and bottom-right elements of the PW are $(2, 8)$ and $(9, 10)$, thereby the PW size of this edge subIFM is 8×3. Similarly, for the example in Figure 6, the initial PW sizes of the corner, edge, and center subIFMs are set to 3×3, 4×3, and 4×4, respectively.

This initial PW initialization method enables subIFM-oriented weight mapping. For each subIFM, the PW size ranges between the kernel size and subIFM size. Equation 1 shows that the corner subIFM size dose not exceed the kernel size. Thus, corner subIFMs have a PW size equal to the kernel size. However, this method incurs large PW sizes for center and edge subIFMs when dealing with large IFMs (e.g., 224×224), causing significant crossbar consumption. Therefore, in resource-limited scenarios, an automated method is essential for selecting the optimal PW size for center and edge subIFMs, which balances crossbar allocation between subIFMs and maximizes overall system performance.

C. Automated PW Selection

Further splitting large PWs for center/edge subIFMs faces two issues: (1) Center/edge subIFMs exhibit uniform N_{ops} values, precluding their use as PW selection guidance. (2) Existing PW selection methods [62], [35] are suboptimal and inefficient due to the lack of global feedback and the enlarged search space (§III-B). To address these issues, we propose an RL-based PW selection method [28].

Why RL? Reinforcement learning (RL) is a powerful method for automated decision-making, particularly effective in scenarios requiring delayed feedback for global optimization. Unlike traditional methods that rely on immediate responses, RL assigns rewards based on the outcome of a sequence of actions [28], [55]. This characteristic aligns well with our problem setting, where each subIFM's PW (parallel-window) size represents an action, and the overall system performance can only be evaluated after all decisions are made.

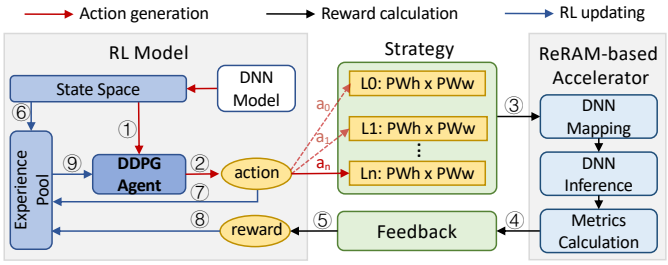


Fig. 9. The RL working process.

By framing the problem this way, our approach leverages RL to achieve global optimization, incorporating real-world hardware metrics directly into the learning process. Since the PW size varies continuously within the range defined by the kernel and IFM sizes, we adopt a Deep Deterministic Policy Gradient (DDPG) agent [43]. This enables fine-grained, continuous control over PW configuration across all convolutional layers, thereby maximizing input parallelism while respecting hardware resource constraints.

Workflow. The RL working process (Figure 9) operates in three phases. *Action generation:* Firstly, the state space provides layer observations (e.g., IFM/kernel sizes) to the RL agent (Step ①). The agent then generates an action (i.e., a PW size) for the center/edge subIFMs of this layer (Step ②). These steps repeat N times until all convolutional layers have received their actions (a_0 – a_n), forming a complete mapping strategy. *Reward calculation:* The mapping strategy is applied on ReRAM-based accelerators for CNN inference (Step ③). Then, the accelerator sends real-world hardware feedback (e.g., latency and energy) back to the RL model to calculate the reward (Steps ④ – ⑤). *RL updating:* The experience pool collects states, actions, and rewards as training samples to update the RL agent (Steps ⑥ – ⑨). The three phases are alternately executed until the reward value converges. To enable efficient RL search, we carefully design each RL component.

Action space. To reduce the search space, we define action a as the number of PW splits rather than specific width or height. This design avoids synchronization issues by using the same a for center/edge subIFMs in a given layer. If the edge subIFM is divided into a splits, the center subIFM is divided into a^2 splits. The action space for layer i is defined as:

$$1 \le a_i \le \frac{L_i}{K_i}, a_i \in \mathbb{Z} \tag{7}$$

where a_i, L_i, and K_i denote the action, the initial PW size of the center subIFM (§IV-B2), and the kernel size, respectively. The new PW size for center/edge subIFMs can be calculated as $PW_i = \lfloor L_i / a_i \rfloor$.

State space. We define a 12-dimensional state vector S to provide observations for the RL agent:

$$S_i = (i, t, W, H, S_{in}, w, C_{in}, C_{out}, K, S, P, a_{i-1}) \tag{8}$$

Detailed feature meanings are cataloged in Table II. All features are directly obtained from the CNN model, except a_{i-1}, which is generated by the RL agent during the previous step.

Reward function. To minimize latency and maximize energy

TABLE II
FEATURES USED IN THE RL STATE SPACE.

No.	Features	Meaning	No.	Features	Meaning
1	i	layer index	7	C_{in}	input channels
2	t	layer type	8	C_{out}	output channels
3	W	width of the IFM	9	K	kernel size
4	H	height of the IFM	10	S	stride length
5	S_{in}	area of the IFM	11	P	padding size
6	w	number of weights	12	a_{i-1}	action of layer $i-1$

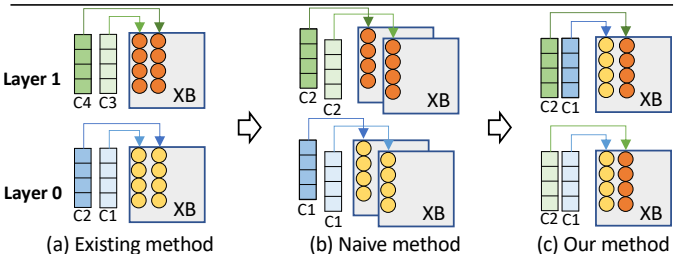


Fig. 10. Three crossbar allocation methods for a two-layer CNN inference. Yellow and orange circles denote weights of two CNN layer. Green and blue squares indicate the input of corresponding subIFMs. Cx denotes input cycles.

efficiency under specific resource constraints, we define the reward R as:

$$R = \begin{cases} \frac{EE}{L} & xb_{consum} \leq XB \\ XB - xb_{consum} & xb_{consum} > XB \end{cases} \quad (9)$$

Here, EE and L represent system energy efficiency and latency, respectively. The term xb_{consum} denotes the crossbar consumption under the current strategy, while XB is the resource limit. EE is defined as throughput (Gops/s) per unit of energy consumption, following the approach in SDK [62].

D. SubIFM-oriented Crossbar Allocation

A key challenge of adopting subIFM-oriented weight mapping in practice is crossbar allocation (§III-B). Figure 10 demonstrates the challenges and our solutions. Consider a simple CNN with two layers, each having the same IFM and kernel size. Their kernels are depicted in yellow and orange, respectively. Since CNN inference is performed layer-by-layer, the computation of the orange layer starts only after the yellow layer completes. We assume that the IFM of each layer is divided into two subIFMs, which have different inputs and correspond to different weights, represented by dark and light colors, respectively.

The existing method of allocating crossbars at the CNN layer granularity may cause weights corresponding to different subIFMs within a layer to be placed on the same crossbar, as Figure 10(a) illustrates. Since each subIFM only performs dot products with its own weights, this method takes two cycles to complete one layer's computation, and thus four cycles for the two-layer CNN. To enable subIFM-level parallel input, Figure 10(b) shows a naive method that allocates crossbars at the subIFM granularity, assigning separate crossbars to weights for different subIFMs. This enables each layer's inference to be completed in one cycle, as computations on different crossbars are performed in parallel and without data dependencies. Thus, the two-layer CNN inference can be completed in just two cycles. However, this approach significantly increases crossbar consumption (i.e., from two to four crossbars).

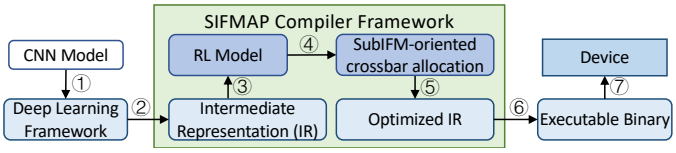


Fig. 11. The SIFMAP compiler framework.

Algorithm 1 Crossbar Allocation Algorithm

Require: *XBSize*: Crossbar size
1: *availableXBList, allocatedXBList, layerList* ← empty Lists
2: **for** $layer$ in $model.layers$ **do**
3: *layerList* ← [] ▷ *layerList* records crossbars of the current layer
4: **for** $block$ in $layer.blocks$ **do** ▷ $block$ denotes a subIFM's weights
5: is_found ← False
6: **for** $xbar$ in *availableXBList* **do** ▷ assign to existing crossbars
7: **if** $block.h \leq xbar.h$ and $block.w \leq xbar.w$ **then**
8: $availableXBList.remove(xbar)$, is_found ← True
9: **break**
10: **end if**
11: **end for**
12: **if** not is_found **then** ▷ assign to a new crossbar
13: $xbar.h \leftarrow \lceil block.h/XBSize \rceil \times XBSize$
14: $xbar.w \leftarrow \lceil block.w/XBSize \rceil \times XBSize$
15: **end if**
16: $layerList.append(\{xbar.h, xbar.w - block.w\})$
17: $allocatedXBList.append(\{block, xbar\})$
18: **end for**
19: $availableXBList.append(layerList)$
20: **end for**
return *allocatedXBList*

To ensure input parallelism and save crossbars, we propose a subIFM-oriented crossbar allocation method in Figure 10(c). It maps the weights for subIFMs within the same layer to different crossbars, while allowing weights from different layers to share the same crossbar. By doing so, this approach completes the entire CNN inference in two cycles, achieving $2\times$ input parallelism than the existing method and reducing 50% crossbar consumption compared with the naive method.

We develop Algorithm 1 to implement this crossbar allocation process. For a given mapping strategy, the algorithm first profiles the crossbar demand for each subIFM in every layer (denoted as block in Line 4). It then assigns crossbars to weights for different subIFMs. If the current subIFM's weights can fit within the idle areas of already allocated crossbars, these weights are mapped to those idle areas (Line 7-10). Otherwise, new crossbars are allocated (Line 12-14). We integrate this algorithm into the RL workflow (Step ③ in Figure 9), enabling it to influence the global decision-making of the RL model by affecting the reward value.

E. SIFMAP Compiler Framework

To deploy SIFMAP in real-world systems, we integrate it into LLVM MLIR, a widely used compiler infrastructure [22]. As Figure 11 shows, the CNN model is initially described using deep learning frameworks (①) and then transformed into an intermediate representation (IR), containing information such as IFM, kernel, and output tensor sizes (②). This IR is fed into a pre-trained RL model for IFM partitioning and automated PW selection (③–④). The compiler generates an optimized IR for mapping weights and performing computations using subIFM-oriented crossbar allocation (⑤). Finally, the SIFMAP compiler uses the LLVM toolchain to generate executable binaries for deployment on PIM devices (⑥–⑦).

Figure 12 contrasts Im2col and SIFMAP IR handling, using the same example as Figure 3 and 6. For simplicity, we only

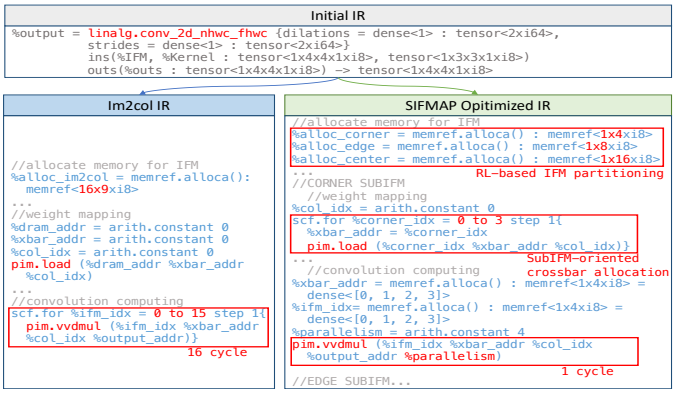


Fig. 12. IRs for Im2col and SIFMAP.

show the results with the corner subIFMs. There are three main differences. First, Im2col flattens the IFM into a 16×9 tensor, while SIFMAP splits it into three types of subIFMs (e.g., 1×4 tensors for corner subIFMs). For the weight mapping, Im2col maps the entire kernel weight to a single crossbar, whereas SIFMAP maps four weight columns to separate crossbars using `pim.load()`. For computation, Im2col runs 16 rounds of vector–vector dot product operations via `pim.vvdmul()`, which multiplies each IFM column by the mapped weights. In contrast, SIFMAP leverages parallelism over four crossbars, substantially boosting hardware efficiency.

V. Evaluation

A. Experiment Setup

Experiment platform. We develop an in-house simulator by modifying MNSIM [65], [64], [50] to estimate the performance of existing works and SIFMAP. MNSIM integrates a CPU, off-chip DRAM and PIM units. Before CNN inference, the CPU produces weight mapping strategies, deploys weights to PIM memory and initializes PIM controller. DRAM serves as large-capacity off-chip storage for storing CNN weights to be deployed and input feature maps (IFMs) to be calculated. After weight deployment, the CPU triggers the PIM to load IFMs from DRAM and start CNN inference. The PIM internal controller schedules the full data and control flow. CNN weights are placed in PIM ReRAM crossbars and intermediate IFMs in multi-level on-chip buffers, eliminating frequent PIM-DRAM data transfers. After the CNN inference is completed, the results are returned to CPU. We adopt the latency and energy efficiency of CNN inference on the PIM as metrics to evaluate different mapping frameworks. The latency is defined as the execution time for one complete CNN inference run on the PIM, excluding the write latency for weight mapping onto ReRAM as well as communication overhead with the CPU and DRAM. Energy efficiency is defined as throughput (Gops/s) per unit energy consumption, following prior work [62]. We adopt each mapping framework for convolutional layers, while other layers rely on MNSIM's built-in methods. For non-MAC operations including nonlinear activations, pooling and normalization, we use the dedicated digital peripheral circuits from MNSIM.

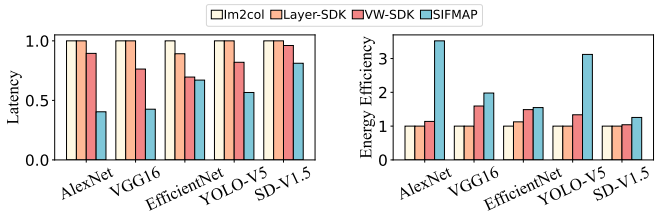
The CNN weights and IFMs are quantized to 8 bits. When processing CNN inference, weights are first separated into positive/negative groups and then bit-sliced across 2-bit quad-level ReRAM crossbars [40]. The inputs are similarly split by sign and decomposed bitwise across multiple computing cycles. Each input slice is converted into an analog signal using 2-bit DACs. The resulting currents are digitized by 8-bit ADCs [5], and the partial sums are accumulated according to their corresponding polarity and bit-position. MNSIM organizes crossbars in a hierarchical structure, including PEs, tiles, and banks. Each ReRAM bank contains 128×128 tiles (16 PEs per tile). The inter-tile and intra-tile communication bandwidths are 20Gbps and 1Tbps, respectively. Other PE configurations follow defaults in Table III [65].

TABLE III
PE CONFIGURATIONS WITH 128 × 128/512 × 512 CROSSBARS.

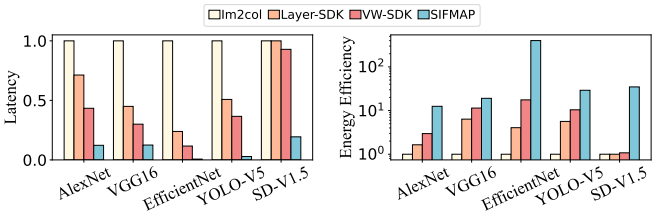
Component	Power (mW)	Area (mm²)	Params.	Spec.
Crossbar	10.145475 / 10.145475	0.377487 / 6.039798	Total	16
ADC	128.000000	0.052800	Total	32
DAC	3.993600	0.000170	Total	512
Input Register	0.004613	0.000350	Size	512B
Output Register	0.000288	0.000022	Size	32B
PE Input Buffer	3.048136	0.040580	Size	8B
Shift Register	0.025920	0.000045	Total	32
Adder	0.000560	0.000064	Total	4
Input Demux	0.049152 / 0.442368	0.000173 / 0.001558	Ratio	1:2 / 1:8
Output Mux	0.248832 / 0.746496	0.000876 / 0.002628	Ratio	32:1 / 128:1

Models and datasets. We select five models of different scales, i.e., AlexNet [20], VGG16 [17], EfficientNet [18], YOLO-V5 [61], and Stable Diffusion-V1.5 (SD-V1.5) [38] as workloads. For AlexNet and VGG16, we use the CIFAR-10 dataset [19], which comprises 60,000 labeled 32×32 images. For larger EfficientNet and YOLO-V5, we use the ImageNet dataset [8], which contains 1.4 million images of various sizes, and rescale the images to 224×224 as the input. For SD-V1.5, we use the LAION-5B [39] dataset with 5.85 billion image-text pairs, and rescale the images to 512×512.

Baselines. We compare SIFMAP with three existing frameworks: The most widely used Im2col [44] adopts kernel-size PWs for IFMs of all convolutional layers. Layer-SDK [62] selectively expands PW sizes for IFMs in certain layers to enhance parallelism. VW-SDK [35] selects rectangular PWs for each IFM using a greedy algorithm. In contrast, SIFMAP enables subIFM-oriented weight mapping, adopts RL-based automated PW selection, and uses subIFM-oriented crossbar allocation. We reimplement these baselines using open-source code [36] and integrate them into MNSIM. We adopt 128×128 and 512×512 crossbar sizes from prior work [62], [35], [40], excluding 32×32 and 64×64 as Layer-SDK and VW-SDK degrade to Im2col under these configurations. For SIFMAP, we train the RL agent for 300 rounds to ensure convergence (reward change below 1% over 20 consecutive rounds [10], [46]), with the optimal strategy as final results. All frameworks are evaluated under the same hardware constraints. The number of DACs and ADCs, together with their resource sharing schemes, are identical across all evaluated frameworks. To conduct a fair comparison among all frameworks under fixed resource constraints, we manually allocate crossbars to each model based on their total parameters. Specifically, the assigned crossbar quantities are listed as follows: AlexNet (12.5K/7.5K), VGG16 (50K/25K), EfficientNet (10M/1.5M), YOLO-V5 (1M/256K), and SD-V1.5 (1.5M/512K), corresponding to the 128 and 512 crossbar sizes respectively. Multiple resource constraint scenarios are examined in the sensitivity analysis (§V-E).



(a) 128×128 crossbars



(b) 512×512 crossbars

Fig. 13. The overall performance of different mapping frameworks.

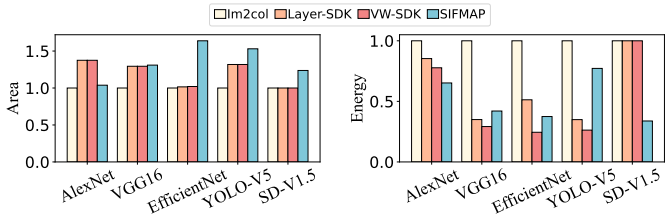


Fig. 14. The area and energy of different mapping frameworks.

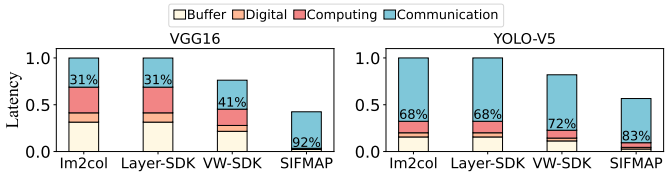


Fig. 15. The latency breakdown of different mapping frameworks.

B. Overall Performance

Latency and energy efficiency. Figures 13 illustrate the latency and energy efficiency of different mapping frameworks across five models under two crossbar sizes, with Im2col normalized to the baseline. SIFMAP achieves optimal latency reduction and energy efficiency gains among all comparisons. Specifically, SIFMAP reduces latency by 1.1–150.1× and improves energy efficiency by 1.1–400× compared to other systems. This highlights the superiority of SIFMAP, which introduces the subIFM-oriented weight mapping and adopts the RL model to determine the optimal mapping strategy for weights of each subIFM based on the whole system feedback. Further, the subIFM-oriented crossbar allocation scheme minimizes crossbar consumption, which provides more available crossbars for weight duplication.

Area and energy overhead. Figure 14 shows the area and energy overhead of different mapping methods on 512×512 crossbars (similar trends for 128×128 results). While SIFMAP maps the weights required by different subIFMs to separate crossbars, its subIFM-oriented crossbar allocation maximizes crossbar reuse, thereby compressing additional area overhead. On average, SIFMAP incurs 1.15× area overhead of VW-SDK, which is acceptable given significant performance gains. Additionally, the average energy overhead of SIFMAP is 1.17× lower than that of VW-SDK. This improvement comes from the significant latency reduction achieved by SIFMAP. We also observe discrepancies in area and energy trends among different models. For AlexNet equipped with dense large convolution kernels, SIFMAP adopts subIFM-oriented weight mapping which maps all weight replicas tightly onto ReRAM crossbars, thus delivering smaller area and lower energy consumption than IFM-oriented weight mapping methods Layer-SDK and VW-SDK. By contrast, for EfficientNet with deeper layers and larger 224×224 IFMs, SIFMAP inevitably introduces extra area and energy overhead. Nevertheless, the subIFM-oriented weight mapping and crossbar allocation strategies of SIFMAP elevate system throughput, yielding considerable improvements in total energy efficiency, as illustrated in Figure 13.

TABLE IV
PW SIZES FOR VGG16 CONVOLUTIONAL LAYERS (Cx) OF DIFFERENT METHODS ON 128×128 CROSSBARS. SIFMAP-CORNER/EDGE/CENTER DENOTES PW SIZES FOR EACH SUBIFM. RATIO REPRESENTS THE PROPORTION OF PW SIZE EQUAL TO KERNEL SIZE.

NO. CONV	C1–C2	C3	C4	C5	C6–C7	C8–C10	C11–13	Ratio
Im2col	3×3	3×3	3×3	3×3	3×3	3×3	3×3	100%
Layer-SDK	3×3	3×3	3×3	3×3	3×3	3×3	3×3	100%
VW-SDK	4×3	3×3	3×3	3×3	3×3	3×3	3×3	84.6%
SIFMAP-corner	3×3	3×3	3×3	3×3	3×3	3×3	3×3	
SIFMAP-edge	3×10	3×3	3×16	3×3	3×5	3×4	3×3	38.4%
SIFMAP-center	10×10	3×3	16×16	3×3	5×5	4×4	3×3	

Latency Breakdown. Figure 15 shows the latency breakdown for VGG16 and YOLO-V5 (similar trends for other models). We divide the total latency into four components: buffer latency covers read/write buffer time; digital latency involves various registers, adders, and joint modules; computation latency includes DAC, ADC, and crossbar computing delays; and communication latency comprises intra-tile and inter-tile data movement time. We observe that the communication latency proportion of SIFMAP rises from 31% to 92% for VGG16 and 68% to 83% for YOLO-V5 compared to Im2col. This is because the subIFM partitioning and allocation of SIFMAP introduce additional data movement. We argue that this overhead is acceptable as SIFMAP still outperforms existing methods in total latency. Besides, SIFMAP reduces buffer latency by 91% and 82% for VGG16 and YOLO-V5, respectively, compared to Im2col. This is attributed to SIFMAP performing computations at subIFM granularity, which minimizes the number of computation cycles and drastically reduces the read/write demands on each buffer.

PW selection results. Table IV presents the PW selection results of different mapping methods on VGG16 using 128×128 crossbars. Similar results are observed for other models. For Layer-SDK and VW-SDK, 100% and 84.6% of PW selection results are equal to the kernel size, indicating that they degrade to Im2col due to crossbar size limitations, as we analyzed in Section II-B. Therefore, both methods achieve similar performance to Im2col, suffering from high latency and low energy efficiency. In contrast, SIFMAP adopts various PWs for different subIFMs of each CNN layer, reducing the ratio of

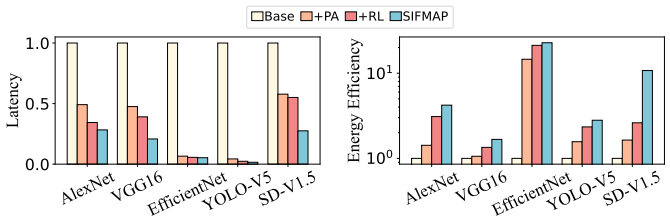


Fig. 16. The performance impact of individual techniques.

TABLE V
PW SIZES FOR VGG16 CONVOLUTIONAL LAYERS (Cx) OF BASE, +PA, +RL AND SIFMAP ON 512 × 512 CROSSBARS.

NO. CONV	C1	C2	C3	C4	C5	C6–C7	C8–C10	C11–13
Base	3×10	4×4	4×4	4×4	4×3	4×3	3×3	3×3
+PA-edge	3×3	3×32	3×16	3×6	3×8	3×5	3×4	3×3
+PA-center	3×3	32×32	16×16	6×6	8×8	5×5	4×4	3×3
+RL-edge	3×10	3×10	3×6	3×6	3×3	3×3	3×4	3×3
+RL-center	10×10	10×10	6×6	6×6	3×3	3×3	4×4	3×3
SIFMAP-edge	3×32	3×32	3×3	3×16	3×3	3×8	3×4	3×3
SIFMAP-center	32×32	32×32	3×3	16×16	3×3	8×8	4×4	3×3

degrading into Im2col to 38.4%. This explains why SIFMAP achieves better latency and energy efficiency in Figure 13. For example, for the weights in the center subIFM of Layer C1, SIFMAP chooses the 10×10 PW size, while Im2col and Layer-SDK use the 3×3 PW size. Due to the benefits of larger PW sizes in enhancing input parallelism, the inference latency of Layer C1 in Im2col, Layer-SDK, VW-SDK and SIFMAP is 8.6ms, 8.6ms, 4.3ms and 21.2µs, respectively.

C. Impact of Individual Techniques

Figure 16 shows the impact of each optimization in SIFMAP. We use 512×512 crossbars and gradually enable each technique. *Base* refers to the state-of-the-art VW-SDK (normalized to one). *+PA* denotes manually determining the PW size using the initial PW selection method (§IV-B2) under resource constraints. *+RL* enables RL-based PW selection (§IV-C), selecting the PW size for each subIFM automatically. SIFMAP extends *+RL* by introducing the subIFM-oriented crossbar allocation method (§IV-D), enabling all optimizations.

We observe that each optimization technique further reduces latency and improves energy efficiency. Specifically, sequentially enabling each optimization achieves average latency reductions of 8.8×, 13.5×, and 18.8×, and improves energy efficiency by 4.1×, 6.1×, and 8.5× compared to *Base*. This demonstrates the necessity of each optimization technique. On average, the contribution ratios of the three techniques across different models are 44%, 19%, and 37% for latency and 25%, 35%, and 40% for energy efficiency, respectively.

To better explain the above results, Table V details PW sizes for VGG16 under 512×512 crossbars across four methods. We omit the results of corner subIFMs as they all remain 3×3. *+PA* achieves fine-grained weight mapping by assigning various PWs to different subIFMs within a single layer. For example, in Layer C2, *Base* uses a single PW size of 4×4, whereas *+PA* assigns 3×3, 3×32, and 32×32 PW sizes to different subIFMs. *+RL* further optimizes PW selection from a global perspective rather than focusing solely on local layers. Under the same resource constraint, *+RL* adjusts the PW sizes of C1–C3 and C5–C7, thus optimizing the latency and energy efficiency of the whole system. SIFMAP enables subIFM-level parallel input by introducing subIFM-oriented crossbar allocation. Compared with *+RL*, it reduces crossbar consumption to free up more crossbars, enabling more layers to adopt larger PWs and thus enhancing performance. Notably, since this allocation occurs at step③ in Figure 9, it influences the reward value, which in turn affects the PW selection in subsequent RL search rounds. For example, SIFMAP expands the PW sizes of the center subIFMs in C1–C2 from 10×10 to 32×32, which effectively reduces latency and boosts energy efficiency.

D. The Advantage of RL Algorithm

To evaluate the efficiency of the RL algorithm, we compare it with two prevalent heuristic algorithms, Genetic Algorithm (GA) [14] and Bayesian Optimization (BO) [9]. All algorithms are implemented using the identical configuration as SIFMAP and undergo the same number of search rounds, i.e., 300, to ensure convergence. Additionally, we also compare RL with the greedy algorithm (GD) used in VW-SDK [35].

Latency and energy efficiency. Figure 17 shows the latency and energy efficiency of different algorithms' search results across five models. RL delivers the lowest latency and highest energy efficiency among four algorithms. Specifically, SIFMAP reduces latency by 1.1–4.7× and improves energy efficiency by 1.1–9.5× compared to GA, BO, and GD. Although GA, BO, and RL all use historical decisions and rewards to adjust future decisions, RL has a key advantage in accessing richer information when making decisions. Namely, RL leverages additional information such as the kernel size, IFM size, and number of weights of each convolutional layer via the state space (§IV-C), which other algorithms lack. This enables the superiority of RL. Besides, RL outperforms GD by focusing on long-term rewards of all layers' decisions rather than the immediate rewards of a single layer, avoiding local optima.

Computing overhead. Since RL training is an offline search process, the computing overhead has no effect on CNN inference performance. Figure 17c demonstrates the search time of four algorithms. RL achieves the lowest latency and highest energy efficiency with search times comparable to or shorter than GA and BO. Although GD achieves the shortest search times on models like AlexNet and VGG16, it struggles with the other three deeper models due to the need to iteratively optimize for each subIFM. The time of training RL model for 300-rounds is 10, 20, 48, 33, and 250 minutes for five models, respectively. We observe that 97% of this time is spent on the simulator to generate the hardware feedback, which will be significantly reduced on a real hardware platform. We consider that the time overhead is acceptable for a contemporary server, as the RL training runs once but yields reusable decisions.

Training efficiency. Figure 18 exhibits the convergence curves for GA, BO, and RL on YOLO-V5 (similar trends for other models). We smooth curves using the classical EMA method [3] with a smoothing parameter of 0.8. GA, BO, and RL start to converge at 266, 273, and 204 rounds, respectively, with best rewards at 175, 182, and 41 rounds. RL outperforms heuristic algorithms in training efficiency as its state vector can provide more contextual information to assist decision-making.

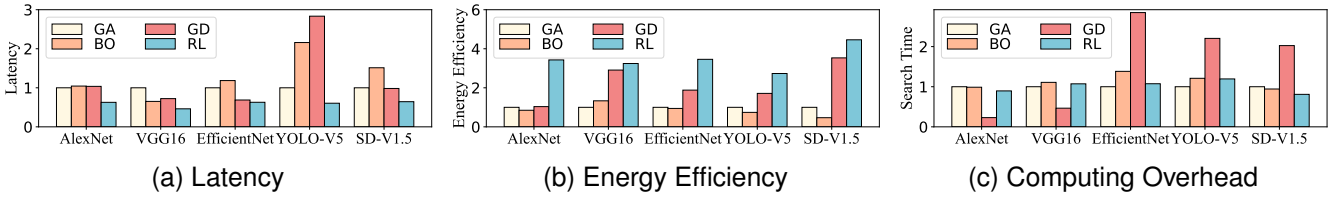


Fig. 17. The comparison of different search algorithms.

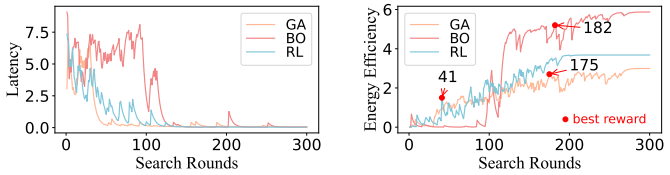


Fig. 18. The convergence curves of different search algorithms.

Scalability on larger models. To evaluate the scalability of RL for larger models, we manually created variants of YOLO-V5 by increasing the number of convolutional modules to 2, 4, 6, and 8 times the original. The results show that the number of training rounds required for RL to converge increased by 1.05×, 1.12×, 1.26×, and 1.42×, respectively. This demonstrates the sublinear scalability of the RL method for larger models.

E. Sensitivity Analysis

To demonstrate the performance of SIFMAP across various scenarios, we evaluate it by adjusting the crossbar sizes and numbers, as well as the learning rates and state space of the RL model. Due to space limits, we present only the results for the state-of-the-art VW-SDK framework (normalized to one) and SIFMAP on YOLO-V5. The other models have similar trends. Furthermore, to explore the robustness and effectiveness of SIFMAP under extreme parameter scenarios, we also conduct experiments with different kernel sizes based on our self-designed CNN models.

Crossbar size. We fix the total number of crossbars and evaluate the performance on four typical crossbar sizes. Figure 19a shows the impact of crossbar sizes (64×64 to 512×512). SIFMAP consistently outperforms VW-SDK across all sizes, achieving an average 5.3× latency reduction and 2.3× energy efficiency increase for YOLO-V5—demonstrating its superiority regardless of crossbar size.

Number of crossbars. We configure crossbars at 512×512 and scale the number of crossbars to 1×, 2×, 4×, and 8× the initial count. Figure 19b confirms SIFMAP's ability to identify optimal mapping strategies under diverse resource constraints, with an average 10.9× latency reduction and 2.7× energy efficiency improvement.

Learning rate. We evaluate different RL learning rates ranging from 10^{-1} to 10^{-5} under the same experimental conditions. Figure 19c illustrates SIFMAP's performance across learning rates from 10^{-1} to 10^{-5}. Despite affecting RL search outcomes, learning rate variations keep SIFMAP consistently superior to VW-SDK—delivering up to 40.1× latency reduction and 2.8× energy efficiency gain. This demonstrates the strong stability of SIFMAP's approach.

State space. We first exclude one feature at a time from the state space (Table II) for RL training. The search results remain unchanged, proving individual features have negligible impact. Then, we group features into three groups and remove one group per trial. Table VI shows RL performs best with all features retained, while removing any group impairs decisions. Kernel-related features are most critical, followed by channel-related and IFM-related ones.

TABLE VI
ABLATION STUDY ON FEATURES. FEATURES MARKED WITH A ✓ ARE INCLUDED IN THE STATE SPACE, WHILE OTHERS ARE NOT.

Features	IFM			Channel		Kernel				Latency ↓ (ms)	Energy Efficiency ↑ (Gops/s/W)
	W	H	S_{in}	C_{in}	C_{out}	K	S	P	w		
All	✓	✓	✓	✓	✓	✓	✓	✓	✓	135.5 (1×)	72.7 (1×)
w/o IFM				✓	✓	✓	✓	✓	✓	257.6 (1.9×)	21.8 (0.16×)
w/o Channel	✓	✓	✓			✓	✓	✓	✓	291.7 (2.1×)	19.9 (0.13×)
w/o Kernel	✓	✓	✓	✓	✓					291.9 (2.1×)	18.8 (0.12×)

Extreme parameters. We manually created six 10-layer CNN models, each adopting a single kernel size (1×1 to 11×11, step 2) and the SAME padding strategy [47]. For 1×1 kernels, SIFMAP degrades to Im2col because all IFM elements have the same operation count, hindering pattern-aware subIFM partitioning. For other kernels, Figure 20 shows SIFMAP achieves an average 2.8× latency reduction and 3.3× higher energy efficiency than Im2col, confirming its robustness and effectiveness with extreme parameters.

VI. DISCUSSION

Scalability to other models. SIFMAP extracts finer-grained computational patterns from applications for refined design and leverages RL to find the optimal solution for the overall system. In convolutional scenarios, we extract discrepancies in IFM operation counts as patterns to enable highly parallel CNN inference via subIFMs. Note that SIFMAP may be limited when dealing with models including multiple 1×1 convolution kernels. This is because 1×1 kernels yield identical IFM operation counts, which hinders pattern extraction and subIFM partitioning. Nevertheless, larger kernel sizes, such as 3×3, 5×5, and 7×7, are predominant in current popular CNN models [20], [17], [18], [61]. For other models (e.g., transformer-based LLMs), new patterns can be defined to enable SIFMAP. For example, in batched LLM inference, requests within a batch have varying output lengths—shorter ones must wait for longer ones to finish before batch completion, causing unnecessary delay [57]. We can extract these length discrepancies as a pattern, group requests with similar predicted output lengths into a subBatch, and use RL to find optimal resource allocation for multiple subBatches, minimizing overall system latency. Similarly, in Mixture of Experts (MoE) models [63], experts exhibit distinct activation frequencies, which leads to imbalanced computation loads. We can extract such disparities in expert activation as a pattern, cluster experts with similar

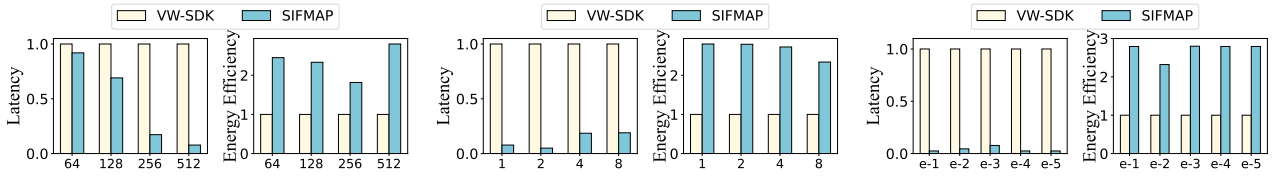


Fig. 19. The results of sensitive analysis on the YOLO-V5 model.

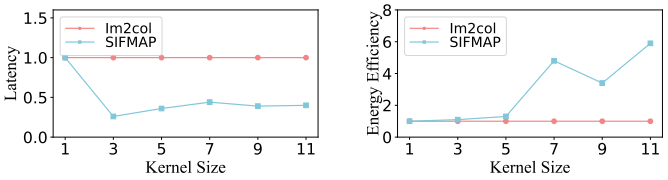


Fig. 20. The performance of SIFMAP on models with extreme kernel sizes.

activation counts into a subGroup, and leverage RL to search for the optimal resource allocation strategy for each subGroup, thereby maximizing overall system performance.

Scalability to other architectures. SIFMAP leverages subIFMs for highly parallel CNN inference, compatible with diverse hardware architectures: MRAM/PCM-based PIM architectures [24], [23] and traditional ones like TPUs and GPUs. Specifically, IFM computation patterns are hardware-agnostic, and the RL model can leverage feedback from these platforms to guide its decision-making process, consistent with the approach used for ReRAM. However, when the hardware platform changes, the subIFM-oriented crossbar allocation scheme requires adjustments to adapt to the new platform. For example, we can implement subIFM-oriented computing core allocation on GPUs to ensure subIFM computing parallelism.

ReRAM robustness for SIFMAP. We choose ReRAM for its efficiency in matrix-vector multiplication along with inherent advantages like non-volatility, high-density, and low power consumption [51], [6], [44]. Since SIFMAP focuses on enhancing CNN throughput using subIFMs, the limitations of ReRAM (e.g., analog noise and device variability) are not solved in this paper. However, we infer that SIFMAP may improve inference accuracy because its expanded parallel weight mappings can increase weight sparsity and provide multiple noise samples for a single weight. Furthermore, extensive research has enhanced ReRAM's robustness—addressing challenges like non-idealities and limited endurance [11], [21], [42]—and this progress is reflected in the emergence of multiple ReRAM chips and commercial products in recent years [60], [7], [4]. These advancements can be integrated into SIFMAP to enhance its robustness for real-world deployment.

VII. Related Work

IFM-oriented methods. We name the existing approaches as IFM-oriented methods, characterized by using the entire IFM as input and mapping weights accordingly. Sub-matrix duplication [31] indiscriminately replicates kernels, enabling parallel input of IFMs to reduce latency. Layer-SDK [62] and VW-SDK [35] employ PWs of varying sizes and shapes for each IFM, mapping weights through shifting and duplicating kernels to enhance performance. However, none of them consider IFM computation patterns, thereby overlooking opportunities to enable finer-grained parallel input. In contrast, SIFMAP leverages these patterns to partition the IFM into subIFMs, enabling subIFM-level parallel input and on-demand weight mapping for efficient performance improvements.

Pruning optimizations. Prune-based mapping methods focus on accelerating CNN inference by exploiting sparsity [37], [30], [33], [34], [49], [54]. They tend to compress model size through weight pruning, thereby impacting CNN inference accuracy. In contrast, SIFMAP maintains model accuracy fidelity. Nevertheless, SIFMAP remains applicable to compressed models. Although pruning alters IFM computation patterns, we can capture these new patterns to partition IFMs into subIFMs, thus improving input parallelism for compressed models.

RL methods for PIM. A variety of RL-based frameworks are proposed to optimize CNN inference on ReRAM-based PIM accelerators. RaQu [32], AUTO-PRUNE [53] and APQ [54] adopt RL agents to conduct layer-wise quantization and pruning, boosting crossbar utilization as well as area and energy efficiency with similar or even higher accuracy. AUTOHET [48] leverages RL to customize heterogeneous crossbar sizes for individual CNN layers, elevating crossbar utilization and reducing overall energy consumption. Unlike the foregoing layer-wise optimization methods, SIFMAP partitions the IFM of each CNN layer into fine-grained subIFMs and leverages an RL agent to derive the optimal weight mapping strategy tailored for subIFMs, which enhances computational parallelism and overall energy efficiency.

VIII. Conclusion

In this paper, we propose SIFMAP, a subIFM-oriented weight mapping framework designed to enable highly parallel CNN inference. SIFMAP first partitions IFMs into subIFMs based on their computation patterns. It then applies an RL method to determine optimal weight mapping strategies tailored to each subIFM, optimizing global performance. Furthermore, SIFMAP introduces a subIFM-oriented crossbar allocation scheme to further boost computational parallelism across subIFMs. Experimental results show that SIFMAP reduces inference latency by $4.8\times$ and improves energy efficiency by $5.2\times$ on average, compared to state-of-the-art mapping methods under equivalent resource constraints.

Acknowledgments

This work was supported in part by the Joint Funds of the National Natural Science Foundation of China under Grant U25A20423, the National Science Foundation of China under Grant 62172361, and the Major Projects of Zhejiang Province under Grant LD24F020012.

REFERENCES

[1] H. Alhichri, A. S. Alswayed, Y. Bazi, N. Ammour, and N. A. Alajlan, "Classification of Remote Sensing Images Using EfficientNet-B3 CNN Model with Attention," *IEEE Access*, pp. 14 078–14 094, 2021.

[2] 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 *Proceedings of the Twenty-fourth International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2019, pp. 715–731.

[3] Z. Cai, A. Ravichandran, S. Maji, C. Fowlkes, Z. Tu, and S. Soatto, "Exponential Moving Average Normalization for Self-Supervised and Semi-Supervised Learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 194–203.

[4] C. Calligaro, "Rad-Hard Resistive Memories," in *Rad-hard Semiconductor Memories*. River Publishers, 2022, pp. 269–307.

[5] H. Chen, X. Zhot, Q. Yu, F. Zhang, and Q. Li, "A¿ 3GHz ERBW 1.1 GS/S 8B Two-Sten SAR ADC with Recursive-Weight DAC," in *2018 IEEE Symposium on VLSI Circuits*. IEEE, 2018, pp. 97–98.

[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*, pp. 27–39, 2016.

[7] P. Corporation, "Panasonic Starts World's First Mass Production of ReRAM Mounted Microcomputers," https://news.panasonic.com/global/press/en130730-2, 2013, accessed 11-10-2024.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-scale Hierarchical Image Database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[9] P. Frazier, "A Tutorial on Bayesian Optimization," *ArXiv*, vol. abs/1807.02811, 2018.

[10] F. G. Ged and M. H. Veiga, "Matryoshka Policy Gradient for Entropy-Regularized RL: Convergence and Global Optimality," *Journal of Machine Learning Research*, pp. 1–52, 2024.

[11] B. Giraud, S. Ricavy, C. Laffond, I. Sever, V. Gherman, F. Lepin, M. Diallo, K. Zenati, S. Dumas, O. Guille, M. Vershkov, A. Bricalli, G. Piccolboni, J.-P. Noel, A. Samir, G. Pillonnet, Y. Thonnart, and G. Molas, "Smart Write Algorithm to Enhance Performances and Reliability of an RRAM Macro," *IEEE Journal of Solid-State Circuits*, 2024.

[12] Y. He, Y. Wang, Y. Wang, H. Li, and X. Li, "An Agile Precision-Tunable CNN Accelerator Based on ReRAM," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–7.

[13] J. Ho, A. Jain, and P. Abbeel, "Denoising Diffusion Probabilistic Models," *Advances in Neural Information Processing Systems*, vol. 33, pp. 6840–6851, 2020.

[14] J. H. Holland, "Genetic Algorithms," *Scientific american*, 1992.

[15] H. Ismail Fawaz, B. Lucas, G. Forestier, C. Pelletier, D. F. Schmidt, J. Weber, G. I. Webb, L. Idoumghar, P.-A. Muller, and F. Petitjean, "Inceptiontime: Finding AlexNet for Time Series Classification," *Data Mining and Knowledge Discovery*, pp. 1936–1962, 2020.

[16] M. Jogin, Mohana, M. Madhulika, G. Divya, R. Meghana, and S. Apoorva, "Feature Extraction Using Convolution Neural Networks (CNN) and Deep Learning," in *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*. IEEE, 2018, pp. 2319–2323.

[17] Karen Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arxiv: 1409.1556*, 2014.

[18] B. Koonce and B. Koonce, "EfficientNet," *Convolutional Neural Networks with Swift for Tensorflow: Image Recognition and Dataset Categorization*, pp. 109–123, 2021.

[19] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," *University of Toronto*, 2012.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification With Deep Convolutional Neural Networks," *Communications of the ACM*, pp. 84–90, 2017.

[21] M. Lanza, H.-S. P. Wong, E. Pop, D. Ielmini, D. Strukov, B. C. Regan, L. Larcher, M. A. Villena, J. J. Yang, L. Goux, A. Belmonte, Y. Yang, F. M. Puglisi, J. Kang, B. Magyari-Köpe, E. Yalon, A. Kenyon, M. Buckwell, A. Mehonic, A. Shluger, H. Li, T.-H. Hou, B. Hudec, D. Akinwande, R. Ge, S. Ambrogio, J. B. Roldan, E. Miranda, J. Suñe, K. L. Pey, X. Wu, N. Raghavan, E. Wu, W. D. Lu, G. Navarro, W. Zhang, H. Wu, R. Li, A. Holleitner, U. Wurstbauer, M. C. Lemme, M. Liu, S. Long, Q. Liu, H. Lv, A. Padovani, P. Pavan, I. Valov, X. Jing, T. Han, K. Zhu, S. Chen, F. Hui, and Y. Shi, "Recommended Methods to Study Resistive Switching Devices," *Advanced Electronic Materials*, p. 1800143.

[22] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[23] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase Change Memory Architecture and the Quest for Scalability," *Commun. ACM*, p. 99–106, 2010.

[24] Y. Li, T. Bai, X. Xu, Y. Zhang, B. Wu, H. Cai, B. Pan, and W. Zhao, "A Survey of MRAM-Centric Computing: From Near Memory to In Memory," *IEEE Transactions on Emerging Topics in Computing*, pp. 318–330, 2023.

[25] Y. Liu, H. Pu, and D.-W. Sun, "Efficient Extraction of Deep Image Features Using Convolutional Neural Network (CNN) for Applications in Detecting and Analysing Complex Food Matrices," *Trends in Food Science & Technology*, vol. 113, pp. 193–204, 2021.

[26] S. Mittal, "A Survey of ReRAM-Based Architectures for Processing-In-Memory and Neural Networks," *Machine Learning and Knowledge Extraction*, pp. 75–114, 2019.

[27] L. Mohammadpour, T. C. Ling, C. S. Liew, and A. Aryanfar, "A Survey of CNN-Based Network Intrusion Detection," *Applied Sciences*, p. 8162, 2022.

[28] S. S. Mousavi, M. Schukat, and E. Howley, "Deep Reinforcement Learning: An Overview," *Springer, Cham*, 2018.

[29] J. Ng, C. Lv, P. Zhao, W. Niu, J. Lin, and Y. Wang, "Open-Source Acceleration of Stable-Diffusion," *arXiv preprint arXiv:2412.05781*, 2024.

[30] J. Park, J. Rhe, and J. H. Ko, "KARS: Kernel-Grouping Aided Row-Skipping for SDK-based Weight Compression in PIM Arrays," in *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2024, pp. 1–5.

[31] X. Peng, R. Liu, and S. Yu, "Optimizing Weight Mapping and Data Flow for Convolutional Neural Networks on Processing-In-Memory Architectures," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1333–1343, 2019.

[32] S. Qu, B. Li, Y. Wang, D. Xu, X. Zhao, and L. Zhang, "RaQu: An Automatic High-Utilization CNN Quantization and Mapping Framework for General-Purpose RRAM Accelerator," in *Proceedings of then 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[33] J. Rhe, K. E. Jeon, and J. H. Kot, "PAIRS: Pruning-AIded Row-Skipping for SDK-Based Convolutional Weight Mapping in Processing-In-Memory Architectures," in *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2023, pp. 1–6.

[34] J. Rhe, K. E. Jeon, J. C. Lee, S. Jeong, and J. H. Ko, "Kernel Shape Control for Row-Efficient Convolution on Processing-In-Memory Arrays," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[35] J. Rhe, S. Moon, and J. H. Ko, "VW-SDK: Efficient Convolutional Weight Mapping Using Variable Windows for Processing-In-Memory Architectures," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 214–219.

[36] J. Rhe, S. Moon, and J. H. Ko, "VW-SDK: Efficient Convolutional Weight Mapping Using Variable Windows for Processing-In-Memory Architectures," https://github.com/djwhsdj/VW-SDK, 2022.

[37] J. Rhe, S. Moon, and J. H. Ko, "VWC-SDK: Convolutional Weight Mapping Using Shifted and Duplicated Kernel with Variable Windows and Channels," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, pp. 408–421, 2022.

[38] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-Resolution Image Synthesis with Latent Diffusion Models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, 2022, pp. 10 684–10 695.

[39] C. Schuhmann, R. Beaumont, R. Vencu, C. Gordon, R. Wightman, M. Cherti, T. Coombes, A. Katta, C. Mullis, M. Wortsman, P. Schramowski, S. Kundurthy, K. Crowson, L. Schmidt, R. Kaczmarczyk, and J. Jitsev, "LAION-5B: An Open Large-scale Dataset for Training Next Generation Image-text Models," *Advances in Neural Information Processing Systems*, pp. 25 278–25 294, 2022.

[40] 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," *ACM SIGARCH Computer Architecture News*, pp. 14–26, 2016.

[41] 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," *ACM SIGARCH Computer Architecture News*, pp. 14–26, 2016.

[42] T. Shahroodi, G. Singh, M. Zahedi, H. Mao, J. Lindegger, C. Firtina, S. Wong, O. Mutlu, and S. Hamdioui, "Swordfish: A Framework for Evaluating Deep Neural Network-based Basecalling using Computation-In-Memory with Non-Ideal Memristors," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1437–1452.

[43] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic Policy Gradient Algorithms," in *Proceedings of the 31st International Conference on International Conference on Machine Learning (ICML)*, 2014, pp. 387—-395.

[44] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 541–552.

[45] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[46] C. J. C. H. Watkins and P. Dayan, "Technical Note: Q-Learning," *Machine Learning*, pp. 279–292, 1992.

[47] S. Wu, G. Wang, P. Tang, F. Chen, and L. Shi, "Convolution with Even-Sized Kernels and Symmetric Padding," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 1194–1205.

[48] T. Wu, S. He, J. Zhu, W. Chen, S. Yang, P. Chen, Y. Yin, X. Zhang, X.-H. Sun, and G. Chen, "AUTOHET: An Automated Heterogeneous ReRAM-Based Accelerator for DNN Inference," in *The 53rd International Conference on Parallel Processing (ICPP '24)*, 2024, p. 10.

[49] Y. N. Wu, P.-A. Tsai, S. Muralidharan, A. Parashar, V. Sze, and J. S. Emer, "HighLight: Efficient and Flexible DNN Acceleration with Hierarchical Structured Sparsity," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2023.

[50] L. Xia, B. Li, T. Tang, P. Gu, P.-Y. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, and H. Yang, "MNSIM: Simulation Platform for Memristor-based Neuromorphic Computing System," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1009–1022, 2017.

[51] Q. Xia and J. J. Yang, "Memristive Crossbar Arrays for Brain-inspired Computing," *Nature Materials*, pp. 309–323, 2019.

[52] P. Yadav, N. Menon, V. Ravi, S. Vishvanathan, and T. D. Pham, "EfficientNet Convolutional Neural Networks-Based Android Malware Detection," *Computers & Security*, p. 102622, 2022.

[53] S. Yang, W. Chen, X. Zhang, S. He, Y. Yin, and X.-H. Sun, "AUTO-PRUNE: Automated DNN Pruning and Mapping for ReRAM-Based Accelerator," in *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2021, pp. 304–315.

[54] S. Yang, S. He, H. Duan, W. Chen, X. Zhang, T. Wu, and Y. Yin, "APQ: Automated DNN Pruning and Quantization for ReRAM-Based Accelerators," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[55] Q. Yao, M. Wang, Y. Chen, W. Dai, Y.-F. Li, W.-W. Tu, Q. Yang, and Y. Yu, "Taking Human out of Learning Applications: A Survey on Automated Machine Learning," *arXiv preprint arXiv:1810.13306*, vol. 31, 2018.

[56] J. You, J.-W. Chung, and M. Chowdhury, "Zeus: Understanding and Optimizing GPU Energy Consumption of DNN Training," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023, pp. 119–139.

[57] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A Distributed Serving System for Transformer-Based Generative Models," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022, pp. 521–538.

[58] G. Yuan, P. Behnam, Y. Cai, A. Shafiee, J. Fu, Z. Liao, Z. Li, X. Ma, J. Deng, J. Wang, M. Bojnordi, Y. Wang, and C. Ding, "TinyADC: Peripheral Circuit-aware Weight Pruning Framework for Mixed-signal DNN Accelerators," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 926–931.

[59] F. Zhang and M. Hu, "Mitigate Parasitic Resistance in Resistive Crossbar-Based Convolutional Neural Networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, pp. 1–20, 2020.

[60] W. Zhang, P. Yao, B. Gao, Q. Liu, D. Wu, Q. Zhang, Y. Li, Q. Qin, J. Li, Z. Zhu, Y. Cai, D. Wu, J. Tang, H. Qian, Y. Wang, and H. Wu, "Edge Learning Using a Fully Integrated Neuro-Inspired Memristor Chip," *Science*, pp. 1205–1211, 2023.

[61] Y. Zhang, Z. Guo, J. Wu, Y. Tian, H. Tang, and X. Guo, "Real-time Vehicle Detection Based on Improved YOLO v5," *Sustainability*, p. 12274, 2022.

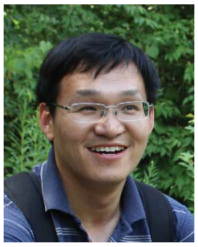
[62] Y. Zhang, G. He, G. Wang, and Y. Li, "Efficient and Robust RRAM-Based Convolutional Weight Mapping With Shifted and Duplicated Kernel," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 287–300, 2021.

[63] X. Zhu, Y. Guan, D. Liang, Y. Chen, Y. Liu, and X. Bai, "MoE Jetpack: From Dense Checkpoints to Adaptive Mixture of Experts for Vision Tasks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[64] Z. Zhu, H. Sun, K. Qiu, L. Xia, G. Krishnan, G. Dai, D. Niu, X. Chen, X. S. Hu, and Y. K. Cao, "MNSIM 2.0: A Behavior-Level Modeling Tool for Memristor-based Neuromorphic Computing Systems," in *Proceedings of the GLSVLSI '20: Great Lakes Symposium on VLSI 2020*, 2020, pp. 83–88.

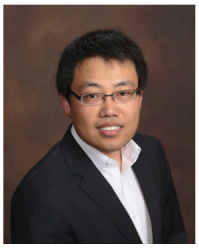
[65] Z. Zhu, H. Sun, Y. Lin, G. Dai, L. Xia, S. Han, Y. Wang, and H. Yang, "A Configurable Multi-Precision CNN Computing Framework Based on Single Bit RRAM," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

Tong Wu is currently working toward the PhD degree with the College of Computer Science and Technology, Zhejiang University, China. Her research focuses on intelligent computing and processing-in-memory for AI.



Shuibing He (Member, IEEE) received the PhD degree in computer science and technology from Huazhong University of Science and Technology, in 2009. He is currently a tenured professor with the College of Computer Science and Technology, Zhejiang University, China. His research areas include intelligent computing, high-performance computing, and memory and storage systems. He is a member of the IEEE and ACM.

Weijian Chen received the PhD degree in computer science and technology from Zhejiang University, in 2025. His research focuses on memory and storage systems for AI.



Xuechen Zhang (Member, IEEE) received the PhD degree in computer engineering from Wayne State University. He is currently an associate professor with the Department of Information Technology, Kennesaw State University, Marietta, GA, USA. His research interests include storage systems and high-performance computing. He is a member of the IEEE and ACM.

Tuo Shi received the PhD degree in materials science from Huazhong University of Science and Technology, in 2017. He is currently a researcher with Zhejiang Laboratory, Hangzhou, China. His research interests include memristor, processing-in-memory and bio-inspired computing.