This paper is published in 2024 Design, Automation and Test in Europe Conference (DATE'24)

PIMSYN: Synthesizing Processing-in-memory CNN Accelerators

Wanqian Li, Xiaotian Sun, Xinyu Wang, Lei Wang, Yinhe Han, Xiaoming Chen*
Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
*Corresponding author: chenxiaoming@ict.ac.cn

***Abstract*—Processing-in-memory architectures have been regarded as a promising solution for CNN acceleration. Existing PIM accelerator designs rely heavily on the experience of experts and require significant manual design overhead. Manual design cannot effectively optimize and explore architecture implementations. In this work, we develop an automatic framework PIMSYN for synthesizing PIM-based CNN accelerators, which greatly facilitates architecture design and helps generate energy-efficient accelerators. PIMSYN can automatically transform CNN applications into execution workflows and hardware construction of PIM accelerators. To systematically optimize the architecture, we embed an architectural exploration flow into the synthesis framework, providing a more comprehensive design space. Experiments demonstrate that PIMSYN improves the power efficiency by several times compared with existing works. PIMSYN can be obtained from https://github.com/lixixi-jook/PIMSYN-NN.**

***Index Terms*—Processing-in-memory, synthesis, neural network accelerators**

I. Introduction

In recent years, processing-in-memory (PIM) has widely been studied for convolution neural network (CNN) inference for performance and power efficiency improvements, which remarkably reduces data accesses between arithmetic and storage components (e.g., [1]–[5]). Compared with traditional CNN accelerators, PIM-based accelerators improve the power efficiency by 2-3 orders of magnitude [4]. Plenty of works implement PIM accelerators based on crossbars composed of ReRAM devices. The crossbar structure can perform an in-situ analog matrix-vector multiplication (MVM) within $\mathcal{O}(1)$ time, which not only eliminates the data accesses for CNN weights but also exploits high parallelism of CNN computation.

PIM-based CNN accelerator design has the following characteristics and challenges which are distinct from conventional CMOS-based CNN accelerators. 1) **Inter-layer pipelining.** Instead of layer-by-layer execution and transferring both weights and activations for every layer in CMOS-based accelerators, a PIM accelerator stores the entire CNN's weights within ReRAM cells and fulfills inter-layer pipelining. PIM accelerators involve a concept of *weight duplication*, which means storing a layer's weights for multiple copies. Duplicating weights enhances computation throughput but greatly expands the design space of both hardware architecture and dataflow scheduling. 2) **Communication bottleneck.** PIM accelerators are typically organized by macros (a macro is typically composed of crossbars and necessary peripheral components), interconnected via a network-on-chip (NoC) or bus. Although PIM alleviates weight access, the demand for communicating activations and intermediate results persists within and between macros. Particularly when weights are duplicated, as computation parallelism increasing, communication emerges as the bottleneck of the accelerator performance. 3) **Energy-intensive peripheral components.** In PIM accelerators, peripheral components, like ADCs and DACs, consume over 60% of the total power [2], playing a critical role in optimizing the power efficiency of PIM architectures.

To develop power-efficient PIM-based CNN accelerators, these factors must be explored to comprehensively optimize power, performance and area (PPA), which greatly amplifies the difficulty of PIM architecture design. Existing PIM-based CNN accelerators are mostly manually designed, which requires both experienced developers and substantial human efforts. It is almost impossible to create PPA-optimized CNN accelerators depending solely on expert experience. Especially when the scale of the design space is extensive, existing works miss a thorough analysis about the mentioned characteristics. Therefore, developing EDA tools which can automatically generate PPA-optimized PIM accelerators is extremely useful.

There are a few works involving the generation and exploration of PIM-based CNN accelerators. Works [6], [7] simply enumerate the architecture parameters (e.g., crossbar size, ReRAM resolution, etc.) assuming a fixed architecture organization and a determined dataflow. AutoDCIM [8] optimizes solely at circuit-level and layout-level for digital PIM architectures, using a template-based generation approach. PIM-HLS [9] places more emphasis on solving memory distribution (SRAM and ReRAM) problems in heterogeneous architectures. But it lacks comprehensive optimization and exploration for homogeneous architectures. The limitation indicates that PIM-HLS cannot be directly applied to the more common practice based on the same device. None of the above succeeds in fulfilling a full-stack synthesis flow to automatically convert CNN tasks into homogeneous PIM architectures, as well as comprehensively considering accelerators' design space.

To address the above challenges, this work makes the following contributions.

- We propose PIMSYN, a full-stack automatic synthesis framework for PIM-based CNN accelerators. PIMSYN realizes a one-click transformation from CNN applications to PIM architectures.

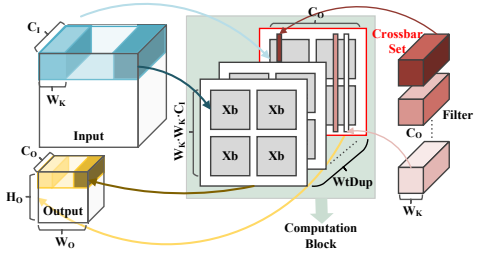

Fig. 1: Crossbar-accelerated convolution computation and weight duplication.

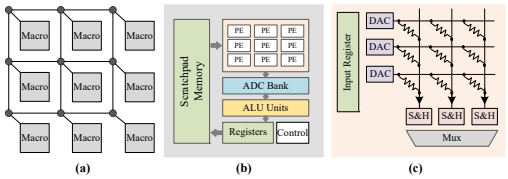

Fig. 2: Architecture abstraction of PIM-based CNN accelerators. (a) Overall architecture. (b) Macro. (c) PE.

- Given the CNN tasks and user-defined power constraint, PIMSYN performs a series of synthesis steps in which design space exploration (DSE) is performed to generate power-efficient PIM accelerators. The design space covers exploration on both dataflows and architectures.
- By defining a set of PIM-friendly intermediate representations (IRs) to express various design choices, PIMSYN greatly expands the accelerator design space and increases the architectural optimization opportunities.
- Experiments demonstrate the superiority of PIMSYN.

II. Preliminary

A. PIM-based CNN Acceleration

Fig. 1 illustrates the scheme of crossbar-based hardware accelerating convolutions. Weights from one kernel filter are programmed into the same column of one or more crossbars. A convolution layer need C_O columns and $W_K \times W_K \times C_I$ rows in total. Due to the limited size of crossbars and resolution of devices, mapping a layer's weights often needs multiple crossbars, denoted as a *crossbar set*. The number of crossbars in one crossbar set can be calculated as

$$set = \left\lceil \frac{W_K W_K C_I}{XbSize} \right\rceil \times \left\lceil \frac{C_O}{XbSize} \right\rceil \times \left\lceil \frac{PrecWt}{ResRram} \right\rceil, \quad (1)$$

where $XbSize$, $PrecWt$ and $ResRram$ are the crossbar size, weight precision and ReRAM cell resolution, respectively. After loading $W_K \times W_K \times C_I$ inputs to crossbars' word lines, a crossbar set can calculate C_O outputs in one step. By duplicating the weights of a layer by $WtDup$ times, $WtDup \times C_O$ outputs can be computed in parallel. This computation process with weights duplicated by $WtDup$ times is denoted as a *computation block*. The inference performance is closely related to the weight duplication factors of layers, namely, $\lceil W_O \times H_O / WtDup \rceil$ steps for completing a layer. If the input activation precision exceeds the DAC resolution, bit-level iterations will be introduced. Each iteration processes input bits equivalent to the DAC precision.

B. Architecture Abstraction of PIM CNN Accelerators

PIMSYN generates PIM-based CNN accelerators based on a PIM-oriented architecture abstraction (i.e., template), as shown in Fig. 2. It is composed of a three-level macro-PE-crossbar hierarchy, where macros are interconnected through a NoC. Different layers are simultaneously executed in macros through pipelining. Communication between layers brings inter-macro synchronization. Each macro consists of a scratchpad memory, a PE array, an ADC bank, ALU components, register files and a controller. In PIMSYN, macros can be configured either identical or specialized for different layers. During CNN inference, PE performs MVM operations and produces analog outputs. A PE includes input registers, DACs, a crossbar, sample and hold (S&H) units and output multiplexers. Results from PE arrays are further converted by the ADC bank and calculated by ALUs which support vector operations (e.g., shift-and-add, pooling, ReLU, etc.). The architecture abstraction is compatible with most previous works, and provides enough flexibility for various implementations. In PIMSYN, all the above components are configurable and parameterizable to enable a comprehensive DSE.

III. PIMSYN Framework Overview

Fig. 3 describes the overview of PIMSYN. The input of PIMSYN includes a CNN model structure described in the ONNX format [10], a total power constraint, and hardware setup parameters (e.g., ReRAM's, ADC's and DAC's latency and power). Through a series of synthesis stages together with the integrated DSE flow, PIMSYN automatically generates the architecture of a PIM-based accelerator with maximized power efficiency. The generated solution also specifies the dataflow scheduling, i.e., when and where each computation task is performed. Since power constraint is an input, maximizing power efficiency is equivalent to maximizing performance. To be mentioned, the CNN model has well been designed, trained, and quantified, and is an input of PIMSYN. Hardware synthesis will not cause any accuracy loss for given CNN algorithms. To ensure that, in PIMSYN, we set the resolution of ADCs to satisfy the minimum resolution requirement according to [2].

The synthesis process of PIMSYN is abstracted into four stages. 1) **Weight duplication.** It decides the weight duplication factor for each layer. It determines layers' parallelism and initializes the inter-layer pipeline. 2) **Dataflow compilation.** It transforms the CNN structural description into the proposed IRs. Dataflow is described by an IR-based directed acyclic graph (DAG), depicting the execution flow and relations between operations. Weight duplication can greatly influence the produced dataflow. 3) **Macro partitioning.** It distributes computation tasks of each layer to respective macros. It also determines data access patterns both within and between macros. 4) **Components allocation.** It assigns functional components inside each macro, such as ADCs, DACs and ALUs.

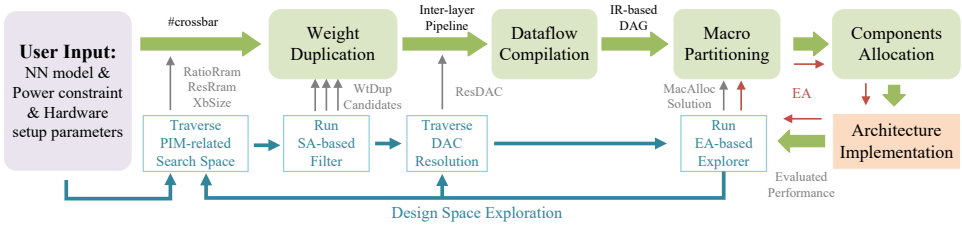

Fig. 3: Overview of PIMSYN framework.

TABLE I: Design space of PIM-based CNN accelerators.

Design variable	Definition	Gibbon [6]	NACIM [7]
$RatioRram$	Ratio of all ReRAMs' power to total power, ranging from 0.1 to 0.4	No	No
$\boldsymbol{WtDup}$	$WtDup^i$, layer i's weight duplication factor	No	No
$XbSize$	Size of crossbar, like 128, 256, 512	Yes	Yes
$ResRram$	ReRAM resolution, like 1, 2, 4	Yes	Yes
$ResDAC$	DAC resolution, like 1, 2, 4	Yes	Yes
$\boldsymbol{MacAlloc}$	$MacAlloc^i$, # of macros assigned to layer i	No	Yes
$\boldsymbol{CompAlloc}$	$CompAlloc^i_j$, # of component j for layer i	No	No

After completing this stage, the accelerator's implementation details are finalized. Among the four stages, the first two stages optimize the dataflow, and the last two optimize the hardware implementation based on the dataflow.

To comprehensively optimize PIM-based CNN accelerators, PIMSYN integrates an architectural exploration flow in the synthesis framework. As shown in Fig. 3, DSE is realized by iteratively exploring through the four synthesis stages. In each iteration, each stage provides a set of design variables for exploration. These design variables collectively determine the generated CNN accelerators. Through continuous iterations and performance evaluations, PIMSYN finds accelerator implementations with optimal power efficiency.

The design space is summarized in Table I. Compared with existing architecture exploration works [6], [7], we expand the design space in several ways. For example, $RatioRram$ and $\boldsymbol{CompAlloc}$[1] reflect the power distribution between different resources and different layers, which greatly affect the accelerator performance. In existing works, they are manually determined. The scale of our defined design space can reach up to 10^{27} for VGG13 [11], making it impossible to traverse all cases. We adopt a simulated annealing (SA) process and an evolution algorithm (EA) to improve the exploration efficiency (see Section IV). Algorithm 1 shows the DSE flow embedded in the synthesis framework. It is a multi-loop based process to iteratively search for the optimal design variables listed in Table I, with embedded SA and EA for boosting the DSE flow.

IV. PIMSYN Synthesis Stages

A. Weight Duplication

1) Problem Definition: Duplicating a layer's weights to more crossbar sets increases parallelism. Different layers are executed in a pipelined way on a PIM accelerator, so limited crossbars should be properly distributed to all layers. The distribution strategy decides weight duplication for each layer and impacts the overall performance. We formulate the decision of weight duplication as a constrained optimization problem as

$$\begin{aligned} \underset{\boldsymbol{WtDup}}{\text{maximize}} \quad & Performance(\boldsymbol{WtDup}) \\ \text{subject to} \quad & \textstyle\sum_{i=1}^{L} \left(WtDup^i \times set^i\right) \leq \#crossbar. \end{aligned} \tag{2}$$

We use $\boldsymbol{WtDup}$ to symbolize the weight duplication strategy. L is the number of layers in the CNN model. $WtDup^i \times set^i$ is the number of crossbars used for layer i, where set^i is calculated by Eq. (1), which depends on two design variables $XbSize$ and $ResRram$.

Solving Eq. (2) also depends on $\#crossbar$ (total number of crossbars) $\#crossbar$ can be determined by the user given total power constraint ($TotalPower$), $RatioRram$, and the power of a single crossbar that depends on $XbSize$ and $ResRram$. As described in Fig. 3, $\#crossbar$ is specified by three design variables, $RatioRram$, $XbSize$ and $ResRram$:

$$\#crossbar = \frac{TotalPower \times RatioRram}{CrossbarPower(XbSize, ResRram)}. \tag{3}$$

The three design variables are explored through traversing them in the PIM-related design space, as shown in lines 3-5 in Alg. 1. They provide $\#crossbar$ for the weight duplication stage and affect the subsequent synthesis stages. The design space of $RatioRram$, ranging from 0.1 to 0.4, is derived from our prior knowledge.

Output: Accelerator implementation
1 $Best\_arch \leftarrow$ NULL;
2 $Best\_perf \leftarrow 0$;
3 **for** $RatioRram \in [0.1, 0.4]$ **do**
4 **for** $ResRram \in \{1, 2, 4\}$ **do**
5 **for** $XbSize \in \{128, 256, 512\}$ **do**
6 $WtDupCandi \leftarrow$ top 30 solutions of SA-based filter;
7 **for** $\boldsymbol{WtDup} \in WtDupCandi$ **do**
8 **for** $ResDAC \in \{1, 2, 4\}$ **do**
9 Generate IR-based dataflow DAG;
10 $\boldsymbol{MacAlloc}, \boldsymbol{CompAlloc} \leftarrow$ best solution of EA-based macro partitioning together with components allocation;
11 Evaluate performance of currently found best solution;
12 Update $Best\_perf$ and $Best\_arch$;

Algorithm 1: Design space exploration flow.

[1]A bold variable denotes a vector containing the corresponding elements of all layers.

2) SA-based Weight Duplication Filter: The size of $\boldsymbol{WtDup}$'s exploration space is the number of all possible positive integer solutions of the problem of Eq. (2). It is typically an astronomical search space. Since $\boldsymbol{WtDup}$ significantly influences the accelerator's performance, solutions that underperform in this stage are unlikely to become the final optimal solutions found by the DSE flow, which offers the opportunity for design space pruning. To shorten the DSE time, we integrate an SA-based filter in this stage to select 30 weight duplication candidates with the lowest energy-function values (line 6), which will be traversed in the future synthesis stages (line 7). Given a specific power constraint, the energy-function of SA is the accelerator's performance. Nonetheless, precisely assessing the accelerator's performance needs the incorporation of subsequent synthesis stages, which complicates the synthesis process. Instead, we devise a straightforward energy-function to represent the performance, which is

$$EnergySA = \underset{i=1,\cdots,L}{\text{stdev}}\left(\frac{W_O^i H_O^i}{WtDup^i}\right) + \alpha \cdot \underset{i=1,\cdots,L}{\text{stdev}}(AccessVolume^i) \quad (4)$$
$$AccessVolume^i = WtDup^i \times (W_K^i \times W_K^i \times C_I^i + C_O^i),$$

where α is an empirical parameter. $AccessVolume^i$ is the data access volume of layer i. Performance-optimal accelerators need to balance computation workload and data access latency for each layer. The SA-based filter selects weight duplication candidates with the smallest $EnergySA$ values that try to balance computation and data access for each layer.

B. Dataflow Compilation

This stage translates the CNN structural description into an IR-based dataflow DAG (line 9), which provides a unified representation of the CNN for the subsequent synthesis stages. The compilation needs the weight duplication strategy from the previous stage and $ResDAC$ as inputs. In the IR-based DAG, nodes represent operations and edges depict the dependencies between operations. IR acts as the interface between high-level algorithms and low-level implementations. Table II lists the defined IRs, including three categories: computation, intra-macro communication and inter-macro communication. The dataflow compilation has three steps.

First, we translate each layer's computation into a set of IRs. As depicted in Fig. 1 and Section II-A, in PIM-based accelerators, parallelism exists in two levels: computation block level and input bit level. Any computation operation of a layer can be denoted by three indices: *layer*, *cnt* and *bit* (see Table II for the meanings of the notations). In PIMSYN, we complement the computation IRs with the three parameters as shown in Table II. We can specify the detailed operations of each layer, when $\boldsymbol{WtDup}$ and $ResDAC$ are provided.

Second, We establish the dependencies between IRs, including inter-layer, inter-block, inter-bit, and inter-operation dependencies, as shown in Fig. 4. Inter-layer dependency is decided by a fine-grained pipeline, which means that a layer can start computation as soon as the previous layer has produced sufficient outputs. Computations of different *computation blocks* and different bits are also performed in a pipelined manner. Inter-operation dependency indicates the order of operations executed within a *computation block*.

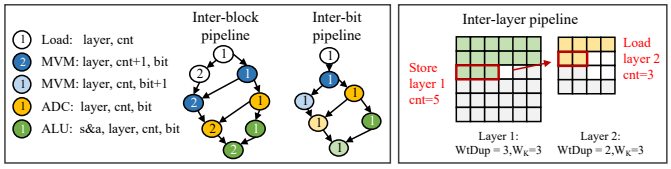

Fig. 4: Dependency relationship between IRs.

Finally, we generate the IR-based DAG. Different from PUMA [5], our IRs can intuitively reflect the computing process of each layer as well as the status of the inter-layer pipeline, which specify the dataflow scheduling. As each IR corresponds to a specific hardware intrinsic, hardware exploration is equivalent to find the optimal resource allocation for IRs. As a result, the performance of synthesized accelerators can be estimated by the depth of the IR-based DAG and the IRs' latencies, which will be discussed in Section IV-D.

C. Macro Partitioning

The high parallelism of PIM-based accelerators brings tremendous pressure to communication. If the resources of a layer are concentrated within a single macro, the communication overhead may be intolerable. Taking ISAAC [2] accelerating conv3 of VGG16 as an example, the weight duplication of conv3 is 64, so the amount of inputs loaded once is about 64KB, taking 20 cycles. To alleviate the stress, we can partition a layer's resources into multiple macros, and they exchange data through the NoC. However, as the number of macros increases, communication between macros becomes complicated, introducing new challenges. The increase in macros also leads to a higher demand for storage resources and peripheral components, especially ADCs. In this stage, we implement an EA-based explorer to search for the macro partitioning solution with the optimal performance, denoted as $\boldsymbol{MacAlloc}$. This stage further supplements communication-related IRs to the dataflow DAG.

1) Rules of Macro Partitioning: PIMSYN supports two scenarios of macro implementations: identical macros across all layers and customized macros for different layers. We stipulate several rules to limit the feasible exploration space of macro partitioning.

a) A layer can occupy one or more macros.

b) Two layers can share the same set of macros.

c) Layer i's crossbars can be partitioned to at most $WtDup^i \times \frac{W_K^i W_K^i C_I^i}{XbSize}$ macros (a macro must have at least one crossbar).

We introduce the opportunity of two layers sharing the same set of macros (rule b). Through macro sharing, two layers reuse the peripheral components, especially ADCs, at different times. We add the macro partitioning choice inspired by the following observation. In PIM-based CNN accelerators, ADCs consume over 60% of total power [2]. To reduce ADC power, most studies adopt intra-layer ADC reuse among columns of a crossbar [2]. However, we find that different layers can

TABLE II: List of intermediate representations.

Category	IR	Parameters	Parameter Explanation
Computation	MVM[a]	layer, cnt, bit, xb_num	cnt: which computation block is currently being calculated
	ADC	layer, cnt, bit, vec_width	layer: which layer the IR belongs to
	ALU	aluop, layer, cnt, bit, vec_width	bit: which bit is currently being calculated in computation block
Intra-Macro Communication	load	layer, cnt, vec_width	xb_num: number of crossbars allocated to the layer
	store	layer, cnt, vec_width	vec_width: length of the operand
Inter-Macro Communication	merge	layer, macro_num, vec_width	aluop: vector arithmetic/logical/non-linear operations
	transfer	layer, src, dst, vec_width	macro_num: number of macros partitioned to the layer

[a] MVM involves DAC and sample-hold. Due to the analog properties, the three operations cannot be divided into different control steps.

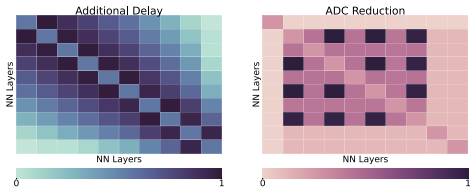

Fig. 5: (a) Normalized delay caused by inter-layer ADC reuse. (b) Normalized number of reduced ADCs after reuse.

Output: Best macro partitioning solution
Randomly initialize the $\boldsymbol{MacAlloc}$ population;
Evaluate each gene;
while $iter < MaxEAIterations$ **do**
    Select parents based on their fitness;
    Apply mutation related to #macros of each layer;
    Apply mutation related to macro-sharing;
    Evaluate new children;
    Insert the children to population;
Return the best solution found;

Algorithm 2: EA-based macro partitioning explorer.

stagger their times for using ADCs, which offers the possibility of inter-layer ADC reuse. Fig. 5 shows ADC reuse between different layers in an example accelerator. When two layers are relatively far apart, ADC reuse hardly brings delay penalty, but it decreases the requirement of ADCs, which can improve power efficiency potentially. Therefore, through searching for the suitable pairs of macro-sharing layers, power efficiency of accelerators can be further optimized.

Although the aforementioned rules can prune part of the $\boldsymbol{MacAlloc}$ design space, it is still impossible to traverse all the design choices. To address that, we propose an EA-based explorer for improving the DSE efficiency.

2) EA-based Macro Partitioning Explorer: In EA, a gene represents a macro partitioning candidate. $\boldsymbol{MacAlloc}$ is encoded by an integer vector, where $MacAlloc^i$ is denoted by $i \times 1000 + \#macro^i$. If layers j and i share the same macros ($j < i$), $MacAlloc^i$ is changed to $j \times 1000 + \#macro^i$. Alg. 2 shows the EA-based explorer, executed on line 10 of Alg. 1.

To efficiently explore macro partitioning, we present two mutation mechanisms in EA: *mutate_num* is to mutate the number of macros assigned to each layer, and *mutate_share* is to change the status of macro-sharing between layers. During the mutation, the generated children always obey the defined rules (Section IV-C1).

We use accelerator performance as the fitness function in the EA-based explorer. As shown in Fig. 3, in each EA iteration, the generated children are passed to the components allocation stage, which will fulfill the architecture implementation and return the performance evaluation result to the EA-based explorer. After multiple iterations, we find a $\boldsymbol{MacAlloc}$ that optimizes the accelerator performance. Meanwhile, $\boldsymbol{MacAlloc}$ specifies the latencies of the communication-related IRs and further completes the IR-based DAG.

D. Components Allocation

This stage accomplishes mapping between IRs and hardware resources. To maximize power efficiency, we need to optimally distribute power among IRs. Different IRs can be assigned to the same physical resource if and only if each pair of these IRs does not have usage conflict. Resource allocation for the *MVM* IR and communication-related IRs are determined before. We develop a heuristic to determine the peripheral components allocation for solving $\boldsymbol{CompAlloc}$.

Peripherals are components outside the PE array, such as the ADC bank and various ALU units (see Fig. 2), which consume a large proportion of total power. The latency of each IR is the ratio of the IR's workload to the amount of its assigned resources. As the pipeline period depends on the most time-consuming step, we model the resource allocation problem as

$$\begin{aligned} \min \quad & \max_{\substack{1\le i\le L,\\ c\in components}} \frac{Wl_c^i}{Freq_c \cdot CompAlloc_c^i}, \\ \text{subject to} \quad & \sum_{i=1}^{L} \sum_{c\in components} P_c \cdot CompAlloc_c^i \\ & = (1 - RatioRram) \times TotalPower, \end{aligned} \tag{5}$$

where $CompAlloc_c^i$ is the number of functional unit c allocated to layer i. Wl_c^i is to the amount of workload that component c needs to perform for layer i. $Freq_c$ and P_c are the frequency and power of component c, respectively. Eq. (5) minimizes the maximum delay under the power limit of the functional units. It is obvious that the best components allocation should balance the delays of all steps. Thus, the solution is ($\forall l \in \{1, \cdots, L\}$ and $\forall p \in components$)

$$\begin{aligned} & (CompAlloc_p^l)_{opt} \times \sum_{i=1}^{L} \sum_{c\in components} \frac{P_c \times Wl_c^i}{Freq_c} \\ & = (1 - RatioRram) \times TotalPower \times \frac{Wl_p^l}{Freq_p}. \end{aligned} \tag{6}$$

V. Experimental Results

We compare the auto-synthsized accelerators with five manually-designed PIM accelerators [1]–[5]. We further compare PIMSYN with the recent model and architectural co-exploration work, Gibbon [6]. The benchmarks include

TABLE III: Part of evaluation and exploration parameters.

Component	Parameters	Values	Power
eDRAM	size, bus_width	64KB, 256b	20.7mW
NoC	flit_size, num_port	32, 8	42mW
ReRAM	crossbar size	128, 256, 512	0.3-4.8mW
	precision	1, 2, 4	
DAC	resolution	1, 2, 4	4-30μW
ADC	resolution	7, 8, $\cdots$, 14	2-54mW

TABLE IV: Peak power efficiency comparison.

	PIMSYN	PipeLayer [3]	ISAAC [2]	PRIME[a] [4]	PUMA [5]	Atomlayer [1]
TOPS/W	3.07	0.14	0.63	0.5	0.84	0.68
Improvement		21.45×	4.83×	6.11×	3.65×	4.51×

[a] Projected to 16-bit quantification as PRIME uses 8-bit quantification.

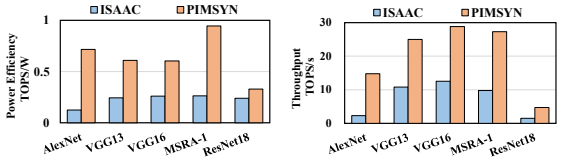

Fig. 6: Effective power efficiency and throughput comparisons with ISAAC.

AlexNet [12], VGG13 [11], VGG16 [11], MSRA [13] and ResNet18 [14] with 16-bit quantification. Some setup parameters in PIMSYN are listed in Table III, together with some exploration parameters during DSE (defined in Table I). Table III only lists key parameters, and other parameters are provided by ISAAC [2] and MNSIM [15]. The synthesized accelerators are evaluated by a cycle-accurate IR-based behavior-level simulator. PIMSYN is implemented in Python and it takes about 4 hours to complete a synthesis process.

A. Comparisons with Manually-Designed Architectures

Peak Power Efficiency. As shown in Table IV, PIMSYN achieves 3.65-21.45× peak power efficiency improvements, 8.11× on average, compared with the five state-of-the-art PIM-based CNN accelerators.

Effective Power Efficiency. It means the achievable power efficiency when running a specific CNN model. Here only ISAAC [2] is compared, because only ISAAC offers detailed parameters to assess the effective power efficiency. Fig. 6 shows that PIMSYN outperforms ISAAC for all five models, with 1.4-5.8× improvements in power efficiency (3.9× on average). The improvement comes from the better power distribution among hardware components. ISAAC has a large portion of power (>80%) consumed by peripheral components, decreasing crossbars' power and computation parallelism.

Throughput. As shown in Fig. 6, PIMSYN achieves 2.30-6.45× (3.4× on average) higher throughput than ISAAC. The reason of the higher throughput is the same as that of the power efficiency improvement.

B. Comparisons with Architectural Exploration Work

Here we compare PIMSYN with Gibbon [6], which fulfills a co-exploration flow of CNN models and architectures. As

TABLE V: Comparisons with Gibbon for CIFAR-10/CIFAR-100 datasets.

		EDP (ms×mJ)	Energy (mJ)	Latency (ms)
AlexNet	Gibbon	0.38/0.38	0.38/0.38	0.99/0.99
	PIMSYN	0.024/0.024	0.119/0.119	0.197/0.197
VGG16	Gibbon	17.22/17.25	2.68/2.68	6.43/6.44
	PIMSYN	7.94/7.95	2.98/2.99	2.66/2.66
ResNet18	Gibbon	4.75/4.76	1.33/1.33	3.58/3.58
	PIMSYN	3.76/3.78	2.34/2.35	1.61/1.61

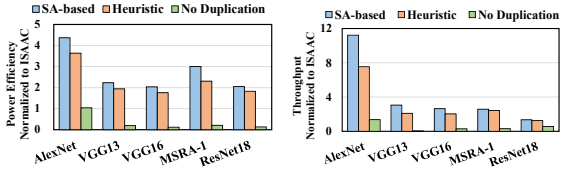

Fig. 7: Power efficiency and throughput improvements brought by different weight duplication methods.

PIMSYN does not involve model exploration, we take CNN models which are trained by Gibbon as inputs. Table V shows that PIMSYN finds better architectures with an average of 56% decrease on energy-delay product (EDP), compared with Gibbon, for CIFAR-10/CIFAR-100 datesets.

C. Effectiveness of Enlarged Design Space

Here we demonstrate the effectiveness of the enlarged design space in PIMSYN, as listed in Table I.

1) SA Selected Weight Duplication: In PIMSYN, $\boldsymbol{WtDup}$ is selected by the SA-based filter. Previous works [2], [3] use a heuristic $\boldsymbol{W_O H_O}$-proportional method (layers' weight duplication factors are proportional to layers' $W_O H_O$). Fig. 7 compares the two methods. PIMSYN achieves 19% power efficiency improvement and 27% throughput improvement. The heuristic decides $\boldsymbol{WtDup}$ based on layers' workloads, which intuitively tries to balance layers' computation latencies. However, it usually requires a large number of crossbars and sufficient bandwidth to support high computation parallelism. In contrast, PIMSYN can be applied to various scenarios, even for power-constrained and bandwidth-limited conditions.

Existing architectural exploration works [6], [7] do not involve weight duplication. In such a case, the power efficiency and throughput are tens of times lower (Fig. 7), indicating the necessity of weight duplication in the designs of PIM-based CNN accelerators and the corresponding workload mapping.

2) Specialized Macro Design: The above results are all for specialized macros. Fig. 8 compares between synthesized accelerators with identical and specialized macros. It show that the specialized macro design realizes 13% power efficiency improvement and 31% throughput improvement. The advantages come from the decrease of the macros' number, which reduces the memory power and inter-macro communication.

3) Inter-layer Macro Sharing: Fig. 9 compares between synthesized accelerators with and without inter-layer macro reuse. Macros in both cases are specialized. Inter-layer ADC reuse makes the effective number of ADCs allocated to the shared layers more than before. It shortens the inference time

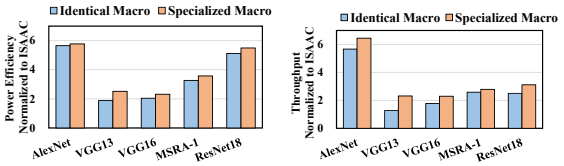

Fig. 8: Power efficiency and throughput improvements brought by specialized macro design.

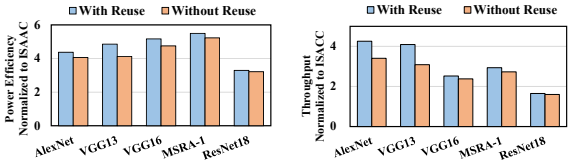

Fig. 9: Power efficiency and throughput improvements brought by inter-layer macro sharing.

if the pipeline period is dominated by ADCs. Inter-layer macro sharing increases 8% and 15% in the overall power efficiency and throughput, respectively.

VI. CONCLUSION

This work develops an automatic synthesis framework for generating PIM-based CNN accelerators, which frees architecture designers from the complexity of manual design. Given CNN models and power constraints, PIMSYN synthesizes the dataflows and architectures of power-efficient accelerators. Compared with previous works, PIMSYN greatly enhances the potential of CNN acceleration by PIM. PIMSYN actually does not rely on the specific device, like ReRAMs. It uses the abstract architecture template that needs some device parameters (e.g., read power and latency). PIMSYN can be used to synthesize any crossbar-based PIM CNN accelerators.

REFERENCES

[1] X. Qiao *et al.*, "Atomlayer: A universal reram-based cnn accelerator with atomic layer computation," in *DAC*, 2018, pp. 1–6.

[2] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *ISCA*, 2016, pp. 14–26.

[3] L. Song *et al.*, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *HPCA*, 2017, pp. 541–552.

[4] P. Chi *et al.*, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ISCA*, 2016.

[5] A. Ankit *et al.*, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *ASPLOS*, 2019.

[6] H. Sun *et al.*, "Gibbon: An efficient co-exploration framework of nn model and processing-in-memory architecture," *IEEE TCAD*, 2023.

[7] W. Jiang *et al.*, "Device-circuit-architecture co-exploration for computing-in-memory neural accelerators," *IEEE TC*.

[8] J. Chen *et al.*, "Autodcim: An automated digital cim compiler," in *DAC*, 2023.

[9] Y. Zhu *et al.*, "Pim-hls: An automatic hardware generation tool for heterogeneous processing-in-memory-based neural network accelerators," in *DAC*, 2023.

[10] J. Bai *et al.*, "Onnx: Open neural network exchange," https://github.com/onnx/onnx, 2019.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015, pp. 1–14.

[12] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, pp. 84–90, may 2017.

[13] K. He *et al.*, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *ICCV*, 2015, pp. 1026–1034.

[14] ——, "Deep residual learning for image recognition," in *CVPR*, 2016.

[15] Z. Zhu *et al.*, "Mnsim 2.0: A behavior-level modeling tool for processing-in-memory architectures," *IEEE TCAD*, pp. 1–1, 2023.