Mapping of CNNs on multi-core RRAM-based CIM architectures

Rebecca Pelke[®], Nils Bosbach[®], Jose Cubero[®], Felix Staudigl[®], Rainer Leupers[®], and Jan Moritz Joseph[®] *RWTH Aachen University, Germany*

Abstract—Resistive random access memory (RRAM)-based multi-core systems improve the energy efficiency and performance of convolutional neural networks (CNNs). Thereby, the distributed parallel execution of convolutional layers causes critical data dependencies that limit the potential speedup. This paper presents synchronization techniques for parallel inference of convolutional layers on RRAM-based computing-in-memory (CIM) architectures. We propose an architecture optimization that enables efficient data exchange and discuss the impact of different architecture setups on the performance. The corresponding compiler algorithms are optimized for high speedup and low memory consumption during CNN inference. We achieve more than 99 % of the theoretical acceleration limit with a marginal data transmission overhead of less than 4 % for state-of-the-art CNN benchmarks.

Index Terms—CNN, RRAM, CIM, weight mapping

I. INTRODUCTION

In recent years, the broad use of convolutional neural networks (CNNs) in computer vision applications yielded an ever-growing demand for efficient architectures to handle these compute- and data-intensive workloads. Due to the massive parallelism and reuse capabilities in CNNs, these applications are not only executed on classical von-Neumann architectures but also on specialized hardware including graphics processing units (GPUs) and tensor processing units (TPUs). Today, CNN accelerators exist in various form factors, from power-efficient edge devices to hyper-scaled compute clusters.

Despite the extensive innovation sparked by the ubiquitous use of CNNs, all these custom architectures suffer from one major performance limitation, namely, moving data from the system's main memory to the compute units, and vice versa [1]. In other words, CNN accelerators suffer from the von-Neumann bottleneck [2]. Novel computing-in-memory (CIM) technologies, such as resistive random access memory (RRAM), promise to tackle this bottleneck by unifying memory and computation unit [3]. These designs offer a significant advantage over CMOS-based designs in terms of memory capacity, device density, and power consumption [4].

Previous works presented accelerator architectures that use RRAM crossbars as matrix-vector multiplication (MVM) units [5]–[8]. These architectures are designed hierarchically to scale from single MVM units to complex multi-core systems. To achieve maximum flexibility and scalability, the MVM units are often embedded in CIM cores, which can communicate with each other over a bus system [6], [7].



Fig. 1. Evaluation framework containing architecture (a) and compiler (b).

The accelerators aim at a weight stationary data flow, i.e., the weights of the CNN are statically assigned to RRAM crossbars [9]. This requires the development of new concepts in the compiler domain. The authors of [10]–[15] investigated the translation of conv2D operations to MVMs. They focused on the mapping of kernel weights to RRAM crossbars. To achieve a high speedup, the kernel weights of one layer must be split across multiple CIM cores for parallel processing. This causes critical data dependencies between cores, which are often neglected. Synchronization techniques are needed to resolve these dependencies. They must be considered in the context of the underlying architecture. The authors of [6] proposed a centrally organized synchronization technique. This scheme requires a high amount of non-RRAM memory.

In our work, we enable efficient, low-overhead parallel execution of CNN layers on multi-core RRAM-based CIM architectures. We use decentralized synchronization methods to minimize memory consumption with marginal data traffic overhead. This includes the following contributions:

- An architecture optimized for efficient, decentralized, and event-based communication.
- Compiler algorithms that achieve more than 99% of the theoretical acceleration limit for conv2D layers.
- A cycle-accurate simulator to analyze the influence of different algorithms and architecture parameters.

Fig. 1 illustrates our evaluation framework. The architecture simulator is used to validate and evaluate the proposed algorithms (see Fig. 1(a)). The specification allows setting different architecture parameters to investigate their influence on the CNN inference. The compiler receives a CNN model and an architecture specification as input and generates code that can be executed on the simulator (see Fig. 1(b)).

This work was funded by the Federal Ministry of Education and Research (BMBF, Germany) in the project NeuroSys (Project Nos. 03ZU1106CA).

^{979-8-3503-2599-7/23/\$31.00 ©2023} IEEE

II. BACKGROUND

A. RRAM crossbars

A RRAM device is a non-volatile emerging memory that stores a conductance value. Multiple RRAM devices can be arranged in crossbar structures to enable in-memory computing [16]. On RRAM crossbars, MVMs can be performed in the analog domain in $\mathcal{O}(1)$ [17]. The weights of the neural network are stored in the crossbar. By applying the input values as voltages, currents are generated that correspond to the result of the MVM. Modern RRAM crossbars have been fabricated in different sizes, e.g., 64×64 [6], 128×128 [7].

B. Weight mapping

Performing MVMs is significantly faster than programming the crossbar cells [5]. If the accelerator provides a sufficient number of crossbars, the weights should therefore be programmed only once to ensure an efficient inference phase. Conv2D layers are well suited for the execution on RRAM crossbars since they can be translated into MVMs and the matrices can be reused multiple times. This has been investigated in previous works. One of the first methods, im2col [10], assigns kernel values to crossbar cells with the densest RRAM cell occupancy. In other approaches, the crossbar is more sparsely packed and kernel values are duplicated to increase the input reuse [11], [13]. Since the im2col scheme requires the least number of RRAM cells, an extended im2col method is used in this work, which splits the kernel values of one layer over several crossbars [14], [15].

C. RRAM-based CIM architectures

Several RRAM-based CIM architectures have been presented [6]–[8], [18]. They aim at efficient and parallel execution of MVMs. Besides different interconnect models, they mainly differ in how autonomously the CIM cores can operate. It ranges from simple MVM units to powerful instruction set architecture (ISA)-based cores [19]. Simple MVM units can be driven and synchronized by a central control unit. Autonomous cores, on the other hand, can execute workloads independently and do not need to be actively controlled. This makes them more flexible and performant. They require a more advanced synchronization procedure, which is addressed in this work.

D. Synchronization techniques

The parallel execution of layers causes critical data dependencies that can lead to incorrect results (see Section IV-A). This can be avoided at the cost of performance loss by executing the critical sections in sequence [13] (see Section IV-B). For parallel execution, synchronization methods are required that need to be supported by the architecture.

The authors of [6] introduced a central synchronization scheme. In their architecture, several *tiles* form the accelerator. A *tile* is structured similar to the architecture in Fig. 1(a) and contains a controller, several CIM cores, and shared memory. The synchronization is solved centrally by extending the shared memory with an attribute buffer. This attribute buffer contains two attributes for each data entry, *valid* and



Fig. 2. CIM core architecture, data flow, instructions, and dimensioning.

count. A memory controller maintains the attributes to ensure the correct exchange of data. In this solution, a significant amount of memory is needed to store the attributes. For 64 kB of data in the shared memory, 32 K attributes are required [6]. We improve on this idea by proposing a decentralized synchronization scheme that requires significantly less memory.

III. ARCHITECTURE

We model a multi-core system as a reference architecture (see Fig. 1(a)). The CIM cores are connected by a bus and use shared memory to exchange their data. In this work, the bus system is used to execute one layer. To be able to execute whole CNNs, the system can simply be duplicated. Architecture parameters, e.g., the number of cores and the size of the crossbars, can be specified variably in our model to investigate their influence on the performance (see Section V-C).

Fig. 2 illustrates the CIM core model including data flow, instructions, and dimensioning. The buffer sizes depend on the matrix size that the MVM unit can process. Cores act as initiators and targets of bus transactions. As initiators, they can, e.g., perform LOAD and STORE operations. As a target, they can receive *config* parameters. The general purpose execution unit (GPEU) can perform arithmetic operations and some activation functions like ReLU and LeakyReLU.

Cores operate in two phases. In the setup phase, the CPU configures the CIM cores. The instructions are loaded and kernel values are programmed into crossbar cells. In the inference phase, the CNN layer is executed. After all calculations are completed, an interrupt is signaled to the CPU.

IV. COMPILER

Our compiler is written in Python to enable simple proofs of concept. It compiles conv2D and dense layers from Tensorflow models and generates a *bin* and a *cfg* file for each layer depending on the architecture specification. The *cfg* file is interpreted by the CPU to configure the CIM cores in the setup phase. The *bin* file is loaded into the shared memory of the CIM cores. It contains an instructions section for each core separately in case not all instructions fit into the instruction memory of the core. It provides placeholders for the input feature map (IFM) and output feature map (OFM) of the layer. In the following, mapping and synchronization techniques

are discussed for the conv2D operation. Dense layers can be treated analogously.

A. Operation remapping

Fig. 3(a) shows the main components of a conv2D layer, i.e., IFM, OFM, and kernels. Fig. 3(b) illustrates the im2col scheme [15]. The unrolled kernels form a matrix, which can then be multiplied by $O_X \cdot O_Y$ unrolled vectors from the IFM.

State-of-the-art CNN layers are often too big to be stored in a single crossbar. The kernel values of one layer have to be split over several crossbars (red lines) [10], [14]. Fig. 3(c) shows the assignment of kernel values to cores. In the setup phase, the CPU loads the IFM to the associated placeholder in the shared memory. Bias values are initially written to the placeholder of the OFM. The GPEU is used to accumulate the bias values and to accomplish the activation function.

We extend the multi-core im2col scheme by assigning two group IDs to each core. All cores sharing the same vertical group (VG) ID operate on the same values of the IFM. All cores sharing the same horizontal group (HG) ID generate partial results for the same values in the OFM that have to be accumulated. In the following, the cores are denoted as $C_{HG,VG}$. While the IFM is read-only, both, read and write accesses are performed on the OFM. Considering $M \times N$ crossbars and (K_Y, K_X, K_Z, K_{NUM}) conv2D kernels (HWIO layout), the total number of needed cores C_{NUM} is

$$C_{NUM} = \underbrace{\left[\frac{K_X \cdot K_Y \cdot K_Z}{N}\right]}_{=:P_V} \cdot \underbrace{\left[\frac{K_{NUM}}{M}\right]}_{=:P_H}$$

The area in the shared memory dedicated to the OFM is reused for the exchange of partial results. This keeps the CIM cores lean since the required buffer sizes and synchronization complexity remain minimal. As a consequence, all cores sharing the same HG ID operate on the same OFM locations in the shared memory. The access must be regulated to avoid race conditions. Hence, a synchronization technique is required to ensure that all partial results are accumulated correctly. This means P_H sets of P_V cores need to be synchronized for $O_{V,NUM} = O_X \cdot O_Y$ different output vectors of size M.

B. Synchronization schemes

All output vectors of the OFM stored in the shared memory can be treated as a resource that may only be owned by one core at any time. Fig. 4 illustrates the proposed synchronization techniques for the example of three conflicting cores $C_{0,0}, C_{0,1}$, and $C_{0,P_V-1} = C_{0,2}$ ($P_H = 1, P_V = 3$) with 12 different output vectors, i.e., 12 different resources. To calculate correctly, each core must have owned each resource once to accumulate its partial results.

In the following, the different parallelization and synchronization schemes are presented. A red *sync* line means that the core that releases a resource notifies (CALL) its successor. That is the core that will receive the resource next. The successor must wait (WAIT) for the notification.



Fig. 3. Translation of a conv2D operation (a) into multiple MVMs with matrix dimension $M \times N$ (b) and distribution of kernel matrix values to cores (c).

1) Sequential synchronization: This scheme is the most basic one. It is used in [12], [13]. Conflicting cores operate sequentially and not in parallel, which eliminates the need for complex synchronization procedures. In the example, $C_{0,0}$ gets all resources first. After it has completed all calculations, the next core, $C_{0,1}$, is allowed to operate. The first core, $C_{0,0}$, accumulates the bias values and the last core, C_{0,P_V-1} , applies the activation function to all output vectors (see Fig. 4(a)).

In the following, we propose two schemes, *linear* and *cyclic* synchronization, to achieve parallel processing, i.e., cores of the same HG can operate in parallel.

2) Linear synchronization: The cores process the output vectors in the same order, starting from $C_{0,0}$ to $C_{0,2}$ in the example. Core $C_{0,0}$ has no predecessor and $C_{0,2}$ has no successor. Core $C_{0,0}$ accumulates the bias values and C_{0,P_V-1} applies the activation function for all output vectors (see Fig. 4(b)). In this case, the number of CALL and WAIT operations is

$$P_H \cdot O_{V,NUM} \cdot (P_V - 1)$$
.

3) Cyclic synchronization: In cyclic synchronization, tasks are distributed as fair as possible among the cores. Each core has exactly one predecessor and one successor. The output vectors are processed cyclically by the cores. The core that first gains access to an output vector accumulates the bias values to its partial results. The core that receives access to an output vector last executes the activation function (see Fig. 4(c)). As a result, the execution of the activation functions and the addition of the bias values are shared equally among all cores. In this case, the number of CALL and WAIT operations is

$$P_H \cdot \left[\frac{O_{V,NUM}}{P_V}\right] \cdot P_V \cdot (P_V - 1).$$



Fig. 4. Sequential computation of OFM without synchronization scheme (a), parallel computation of OFM with linear (b) and cyclic (c) synchronization for conflicting cores $C_{0,0}$, $C_{0,1}$ and $C_{0,2}$, pseudo instructions for parallel computation of a conv2D operation (d).

C. Sequence number

We extend each core with a single register to enable parallelization and synchronization on the architecture side. That is illustrated in Fig. 4(b) and Fig. 4(c). This register contains a sequence number (SEQ_NR) which is writable for other cores. The initial value is 0. A CALL operation increments the sequence number of the successor core (blue line). When executing a WAIT operation, the core waits for its sequence number to reach at least a certain value. Fig. 4(d) shows the pseudo instructions. Three cases are distinguished. In the first case, the core has no predecessor for the output it is working on. In the second case, the core has both, a predecessor and a successor. The last case describes the scenario in which a core is the last one operating on an output.

V. RESULTS

We evaluate our proposed parallelization techniques in terms of speedup gain and overhead costs using the conv2D layers of Mobilenet [20] and ResNet-18 [21]. Those layers are also found in other CNNs. The synchronization methods do not affect the accuracy of the CNNs, which is therefore not examined here. The conv2D layers are compiled for different architecture parameters and synchronization schemes to investigate their effects on the performance.

A. Architecture simulator

Compiled layers are executed on our SystemC/TLM-2.0based simulator to verify the compiler concepts and algorithms. To obtain realistic latency values and enable architectural exploration on a high abstraction level, the TLM-2.0 non-blocking interface is used in combination with the approximately-timed coding style [22]. We use a multiinitiator-multi-target AXI4 bus interconnect [23]. The AXI4 bus protocol [24] supports burst transactions and out-oforder transaction completion, which are beneficial features for data-intensive and highly parallel workloads. We capture relevant data during runtime to evaluate the impact of different architecture parameters and synchronization methods on the performance [25].

B. Performance speedup

To evaluate the parallelization and synchronization methods, we examine the speedup of the linear (Fig. 4(b)) and cyclic schemes (Fig. 4(c)). The speedup always refers to the latency of the corresponding sequential version [12], [13] (Fig. 4(a)).

$$S_{LINEAR} = \frac{t_{SEQUENTIAL}}{t_{LINEAR}}, S_{CYCLIC} = \frac{t_{SEQUENTIAL}}{t_{CYCLIC}}$$

The variable t_X denotes the latency of the inference of a conv2D layer using scheme X. An upper bound for the maximum achievable speedup is P_V , i.e., all conflicting cores run in parallel without synchronization overhead (see Section IV-B).

Fig. 5 shows the speedup of the linear (blue) and the cyclic synchronization method (red) for different conv2D layers of Mobilenet. The shapes of the used layers (Layer #) are listed in Table I. The crossbar dimensions are 32×32 and 64×64 . This, in combination with the kernel shape of the layer, determines the number of needed cores. The upper bound for the maximum achievable speedup (P_V) is indicated by the dashed lines. The speedup increases when reducing the crossbar size. A reduction from 64×64 to 32×32 crossbars results in a speedup of at most $2 \times$ referred to the corresponding sequential scheme. Up to $4 \times$ more cores are required which increases the bus utilization and synchronization complexity. This means that higher speedups can be achieved at the cost of higher numbers of cores and larger bus widths.

Fig. 5 also reveals the speedup which can be achieved for conv2D layers depending on the bus width and crossbar dimension. The figure demonstrates that the speedup limit can be reached even for small bus widths (4B) when the total number of cores is small (\leq 32). Using a large bus width of 32 B, up to 128 cores can operate in parallel. Beyond this, the speedup limit cannot be reached. Reaching the speedup limit for sufficiently high bus widths proves that the synchronization presented in this work does not cause long waiting times. The highest speedup that is achieved is $16 \times$ for layer 5. The speedup of the cyclic method is slightly higher compared to the linear method because the instructions are distributed more evenly among the cores (see Section IV-B). Thus, the linear



Fig. 5. Speedup vs. maximum achievable speedup (dashed lines) of the linear and cyclic synchronization for different layers, crossbar dimensions, and bus widths. The number of cores depends on the layer and crossbar dimension. It refers to 32×32 crossbars (left entry) and 64×64 crossbars (right entry).

method should be preferred due to its simple implementation.

D. Area overhead

C. Bus width

We previously demonstrated that the synchronization methods are very efficient as the speedup limit can almost be attained. However, this limit can only be achieved when the bus is sufficiently wide which prevents it from becoming a bottleneck. Fig. 6 shows to which extent the speedup limit can be reached depending on the crossbar dimension, bus width, and the number of cores. Each line represents one combination of crossbar dimension and bus width. For every combination, conv2D layers from the Mobilenet and ResNet-18 architecture were compiled and simulated. The data from Fig. 6 can be used to determine two things, the appropriate number of cores for a given crossbar dimension and bus width or a reasonable bus width for a given number of cores and crossbar dimension.

In general, the smaller the bus width, the lower the number of cores the bus can handle without becoming a bottleneck. For small bus widths (4 B, red line), only a maximum of 16 cores are worthwhile to achieve more than 90% of the speedup limit, with 64 B (blue line) the system can contain up to 512 cores. If the crossbar dimension is halved, i.e., the number of required cores is quadrupled, then the bus width should at least be doubled to achieve similar performance. For a given number of cores, Fig. 6 can be used to determine a suitable combination of crossbar dimension and bus width.

TABLE I Excerpt from Mobilnet's conv2D Layers

#	kernel shape	matrix shape	input shape			
1	$1 \times 1 \times 128 \times 128$	128×128	$56 \times 56 \times 128$			
2	$1 \times 1 \times 128 \times 256$	256×128	$28 \times 28 \times 128$			
3	$1 \times 1 \times 256 \times 256$	256×256	$28 \times 28 \times 256$			
4	$1 \times 1 \times 256 \times 512$	512×256	$14 \times 14 \times 256$			
5	$1 \times 1 \times 512 \times 512$	512×512	$14 \times 14 \times 512$			
6	$1 \times 1 \times 512 \times 1024$	1024×512	$7 \times 7 \times 512$			
7	$1\times1\times1024\times1024$	1024×1024	$7\times7\times1024$			

Apart from the performance, the overhead caused by synchronization needs to be considered. A major advantage of the methods presented in this work is that synchronization can be realized by simple register accesses. Only one register per CIM core is needed. Assuming that one of the 32 K attributes from [6] requires one byte of memory (see Section II-D), our approach saves at least 87.5% of the synchronization memory, since with a maximum of 1024 cores and 4 B per register, only 4 kB of memory is required.

E. Synchronization overhead

Synchronization requires additional operations. In contrast to the WAIT operation, the CALL operation must be transferred over the bus, which increases bus traffic. The smaller the crossbar size, the more cores are needed, and the more CALL operations have to be executed.

Fig. 7 shows that the bus traffic caused by CALL operations is small compared to the data values transferred over the bus. For CALL operations with a size of 4 B and data values with a size of 1 B, the overhead is less than 4% when using 32×32



Fig. 6. Speedup divided by speedup limit of cyclic synchronization scheme for different layers, bus widths, and crossbar dimensions.

TABLE II NUMBER OF CALL INSTRUCTIONS AND LOAD/STORE VALUES

#	32x32 XBar				64x64 XBar			128x128 XBar				
	Cores[#]	Load[val]	Store[val]	Calls[#]	Cores[#]	Load[val]	Store[val]	Calls[#]	Cores[#]	Load[val]	Store[val]	Calls[#]
1	16	2809856	1605632	37632	4	1204224	802816	6272	1	401408	401408	0
2	32	1404928	802816	18816	8	602112	401408	3136	2	200704	200704	0
3	64	3010560	1605632	43904	16	1404928	802816	9408	4	602112	401408	1568
4	128	1505280	802816	21952	32	702464	401408	4704	8	301056	200704	784
5	256	3110912	1605632	47040	64	1505280	802816	10976	16	702464	401408	2352
6	512	1555456	802816	23520	128	752640	401408	5488	32	351232	200704	1176
7	1024	3161088	1605632	48608	256	1555456	802816	11760	64	752640	401408	2744

crossbars, less than 2% when using 64×64 crossbars, and less than 1% when using 128×128 crossbars.

Table II shows the absolute number of LOAD and STORE operations. Note that the number of loaded values is greater than the number of values in the IFM and the number of stored values is greater than the number of values in the OFM. The reason for this is that the exchange of partial results and the loading of bias values are also counted. Some input values are loaded multiple times because the same input values are multiplied by different kernel values (see [13]).



Fig. 7. Bus traffic caused by CALL operations (4 B per operation) in relation to transferred data (1 B per data value). VI. CONCLUSION

This paper proposes efficient, low-overhead synchronization techniques to enable the parallel execution of single layers on RRAM-based multi-core CIM architectures. We introduce an architecture that supports synchronization and data exchange in a decentralized and event-based manner. The synchronization mechanisms require significantly less memory compared to the state of the art. On the compiler side, we generate code for different architecture setups and evaluate them on a simulator. By exploiting the synchronization mechanisms of the architecture, we achieve more than 99% of the theoretical acceleration limit for conv2D layers of state-of-the-art CNNs with less than 4% additional bus traffic.

This work contributes to understanding the challenges of mapping CNNs to multi-core CIM systems. The presented techniques can be used as building blocks for compilers to enable parallel inference of CNN layers. As a future step, data dependencies between different layers must be considered to enable full system-level integration.

REFERENCES

[1] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

- [2] X. Zou *et al.*, "Breaking the von neumann bottleneck: architecture-level processing-in-memory technology," *Sci. China Inf. Sci.*, 2021.
- [3] Y.-F. Chang *et al.*, "Memcomputing (memristor+ computing) in intrinsic sio x-based resistive switching memory: Arithmetic operations for logic applications," *IEEE (T-ED)*, vol. 64, no. 7, pp. 2977–2983, 2017.
- [4] J. S. Vetter and S. Mittal, "Opportunities for nonvolatile memory systems in extreme-scale high-performance computing," *CiSE*, 2015.
- [5] W. Wan et al., "A compute-in-memory chip based on resistive randomaccess memory," Nature, vol. 608, no. 7923, pp. 504–512, 2022.
- [6] A. Ankit et al., "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in ASPLOS XXIV, 2019.
- [7] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," ACM SIGARCH Computer Architecture News, vol. 44, no. 3, pp. 14–26, 2016.
- [8] P. Chi et al., "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," ACM SIGARCH Computer Architecture News, vol. 44, no. 3, pp. 27–39, 2016.
- [9] X. Liu *et al.*, "Fpra: a fine-grained parallel rram architecture," in 2021 ISLPED. IEEE.
- [10] K. Yanai et al., "Efficient mobile implementation of a cnn-based object recognition system," in *Proceedings of the 24th ACM international* conference on Multimedia, 2016, pp. 362–366.
- [11] Y. Zhang *et al.*, "Efficient and robust rram-based convolutional weight mapping with shifted and duplicated kernel," *IEEE TCAD*, vol. 40, 2020.
- [12] J. Rhe *et al.*, "Vw-sdk: efficient convolutional weight mapping using variable windows for processing-in-memory architectures," in *DATE*. IEEE, 2022.
- [13] J. Rhe *et al.*, "Vwc-sdk: Convolutional weight mapping using shifted and duplicated kernel with variable windows and channels," *IEEE JETCAS*, 2022.
- [14] S. Negi et al., "Nax: neural architecture and memristive xbar based accelerator co-design," in Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC), 2022, pp. 451–456.
- [15] A. Agrawal *et al.*, "X-changr: Changing memristive crossbar mapping for mitigating line-resistance induced accuracy degradation in deep neural networks," *arXiv preprint arXiv:1907.00285*, 2019.
- [16] W. Cao et al., "Neural-pim: Efficient processing-in-memory with neural approximation of peripherals," IEEE Transactions on Computers, 2021.
- [17] B. Li *et al.*, "Rram-based analog approximate computing," *IEEE TCAD*, vol. 34, no. 12, pp. 1905–1917, 2015.
- [18] L. Song et al., "Pipelayer: A pipelined reram-based accelerator for deep learning," in 2017 IEEE HPCA. IEEE, 2017, pp. 541–552.
- [19] S. Mittal, "A survey of reram-based architectures for processing-inmemory and neural networks," *Machine learning and knowledge extraction*, vol. 1, no. 1, 2018.
- [20] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv*:1704.04861, 2017.
- [21] K. He et al., "Deep residual learning for image recognition," in IEEE CVPR, 2016, pp. 770–778.
- [22] J. Aynsley et al., "Osci tlm-2.0 language reference manual," Open SystemC Initiative (OSCI), p. 15, 2009.
- [23] ARM, "Arm amba tlm 2.0 library developer guide," 2019.
- [24] ARM, "Amba axi and ace protocol specification," 2013.
- [25] N. Bosbach et al., "Nistt: A non-intrusive systemc-tlm 2.0 tracing tool," in 2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC). IEEE, 2022.