CIM-MLC: A Multi-level Compilation Stack for Computing-In-Memory Accelerators

Songyun Qu*
Shixin Zhao*
Institute of Computing Technology,
CAS, University of Chinese Academy
of Sciences
Beijing, China
qusongyun18z,zhaoshixin22s@ict.ac.cn

Bing Li
Capital Normal University
Beijing, China
bing.li@cnu.edu.cn

Yintao He
Institute of Computing Technology,
CAS, University of Chinese Academy
of Sciences
Beijing, China
heyintao19z@ict.ac.cn

Xuyi Cai
Institute of Computing Technology,
CAS, University of Chinese Academy
of Sciences
Beijing, China
caixuyi18s@ict.ac.cn

Lei Zhang
State Key Lab of Processors, Institute
of Computing Technology, Chinese
Academy of Sciences
Beijing, China
zlei@ict.ac.cn

Ying Wang†
State Key Lab of Processors, Institute
of Computing Technology, Chinese
Academy of Sciences
Beijing, China
wangying2009@ict.ac.cn

Abstract

In recent years, various computing-in-memory (CIM) processors have been presented, showing superior performance over traditional architectures. To unleash the potential of various CIM architectures, such as device precision, crossbar size, and crossbar number, it is necessary to develop compilation tools that are fully aware of the CIM architectural details and implementation diversity. However, due to the lack of architectural support in current popular open-source compiling stacks such as TVM, existing CIM designs either manually deploy networks or build their own compilers, which is time-consuming and labor-intensive. Although some works expose the specific CIM device programming interfaces to compilers, they are often bound to a fixed CIM architecture, lacking the flexibility to support the CIM architectures with different computing granularity. On the other hand, existing compilation works usually consider the scheduling of limited operation types (such as crossbar-bound matrix-vector multiplication). Unlike conventional processors, CIM accelerators are featured by their diverse architecture, circuit, and device, which cannot be simply abstracted by a single level if we seek to fully explore the advantages brought by CIM.

*Both authors contributed equally to this research.
†Corresponding author.

Therefore, we propose CIM-MLC , a universal multi-level compilation framework for general CIM architectures. In this work, we first establish a general hardware abstraction for CIM architectures and computing modes to represent various CIM accelerators. Based on the proposed abstraction, CIM-MLC can compile tasks onto a wide range of CIM accelerators having different devices, architectures, and programming interfaces. More importantly, compared with existing compilation work, CIM-MLC can explore the mapping and scheduling strategies across multiple architectural tiers in CIM, which form a tractable yet effective design space, to achieve better scheduling and instruction generation results. Experimental results show that CIM-MLC achieves 3.2× inference speedup on average compared to prior CIM-oriented compilation work.

1 Introduction

Advancements in computer performance have been trapped by the famous "*memory wall*" problem for decades [45]. The advent of data-intensive workloads, like DNNs [32], further exacerbates this problem. As a promising technology to combat the memory wall, computing-in-memory (CIM) has garnered significant attention for its capability of reducing frequent data movement and parallelizing multiply-and-accumulate (MAC) operations. A variety of CIM-based

architectures emerge, achieving significantly improved computing efficiency for DNN applications over traditional architectures [4, 6, 11, 13, 18, 39, 42].

To fully exploit CIM, an efficient compilation tool that bridges the DNN scope and various CIM hardware is needed. However, the development of CIM accelerators shows the following trends that pose thorny challenges to the design of a general compilation tool: **(1) The diversity of CIM memory devices.** Unlike the traditional computing paradigm, CIM hinges on memory devices to perform computations, in which the attributes of memory devices, like their types and precision levels, exert a substantial influence on the feasible scheduling space within a CIM compilation framework. For example, when comparing SRAM[26] and ReRAM, although both have similar latency for read operations, the cost of writing data is considerably higher in ReRAM [3]. Thus, SRAM-based CIM supports flexible data read and write updates on CIM memory [6], while ReRAM-based CIM usually assumes that weights are frozen in the crossbar, avoiding the penalty of frequent writes [13, 39]. These factors seriously impact the scheduling strategy and space of a CIM compiler.

(2) The diversity of CIM architectures. In contrast to the common tiled PE-arrays of typical DNN accelerators [12], the architecture of CIM exhibits a diverse and distinct organization[30], like the structure of an individual computing unit, the number of crossbars within each computing unit, crossbar size, *etc.* Consequently, the optimization space of the CIM compiler becomes much larger. The CIM compiler must possess an understanding of the manifold forms within CIM architecture when scheduling and mapping DNN operations onto CIMs.

(3) The diversity of CIM programming interface. CIM architectures have rich hierarchies, and the manner in which they are exposed to programmers varies across different chip designs. For example, some CIMs support fined-grained row-wise operations to realize more general-purpose arithmetic except convolutions and matrix multiplication [27]. To fully exploit CIM architecture across different applications, researchers have developed the CIM-oriented programming interface [4, 19]. However, the existing programming interface is deeply bound to a specific CIM design. With different programming interfaces, the granularity of the elementary computing unit that the scheduler can manage during optimization varies. A fine granularity leads to a more complex scheduling space, which increases the difficulty of compilation optimization.

Therefore, the key challenges to developing a general-purpose CIM compiler are: *1. How to perform multi-dimensional hardware abstraction on CIM to enhance the generality of the compiler? 2. How to generate the optimal mapping and scheduling for different CIMs that expose different operation levels?*

Unfortunately, existing works are difficult to adapt to the diversity of CIM accelerators, and lack optimization considerations for the specific computing granularity in different

CIMs, resulting in inferior performance and even failure of compiling. Some works manually deployed the model on CIMs [39] with customized mapping and scheduling policies that are hard to generalize to other CIMs. Some works proposed CIM-oriented compilation tools [2, 17, 22]. However, they are insufficient to tackle the above challenges. Most of the CIM compilers neglect the generality in their design. For example, Ambrosi *et al.* [2] propose a compilation tool that schedules matrix-vector computation (MVM) on a ReRAM-based architecture, but its performance degrades when the CIM architecture and computing granularity change. Meanwhile, they may also fail to fully explore the scheduling space of DNN operators at the corresponding computing level. For example, Han *et al.* [22] design a compilation tool to deploy DNNs on the ISAAC architecture [39] via an MVM-grained programming interface, but its optimization strategy stays at the computing graph level neglecting the opportunity to fine-control the crossbar resource allocation and MVM operation sequencing for better results.

In this work, we propose a multi-level compilation stack for CIM accelerators to achieve more versatile and efficient compilation. In particular, we first introduce a three-tier architecture abstraction of CIM hardware, from the crossbar tier to the chip tier. Each tier has corresponding hardware architecture parameters, which can be designated to describe a particular CIM accelerator. Then, based on the hardware abstraction of CIM accelerators, we propose the computing mode abstraction with computing granularity ranging from fine to coarse to suit various programming interfaces. At different computing modes, we abstract the basic operation types of a CIM accelerator as meta-operator sets, specifically supported operators in the CIM like memory row read/write, and MVM operator with crossbars. We can then use the meta-operator sets to represent the computation process of DNNs on CIM accelerators.

Within the provided software and hardware space, we propose a multi-level scheduling approach to optimize mapping and scheduling for different computing modes. According to the computing mode given by the target CIM, our policy progressively optimizes computations from coarse-level computing graphs to fine-level vector computing operations and generates the corresponding meta-operator flow. Compared to existing compilation methods that optimize schedules at a single computational level, such as the DNN computing graph, our approach offers a more holistic optimization perspective, resulting in higher computing efficiency.

The contributions of this work are summarized as follows:

- This paper proposes a compilation stack, CIM-MLC, which enables automatic compilation for various CIM accelerators. In CIM-MLC, we propose a hardware abstraction of hierarchical architecture and computing mode of CIM architecture to support a wide range of CIM accelerators.

- Then, we propose a multi-level DNN scheduling approach, which realizes the flexible DNN operators mapping and scheduling based on the computing granularity exposed by specific CIM accelerators. We conduct thorough optimization at each abstraction level to generate the operation flow. The proposed multi-level optimization flow covers a much broader scheduling space than the previous graph or MVM-level scheduling but also avoids the intractability faced by single-level fine-grained scheduling.

- Compared with existing compilation work [22], the proposed CIM-MLC can improve the inference speed by up to 3.2×. We also verify the proposed compilation stack on existing CIM accelerators, PUMA [4], CIM work [27] and CIM work [29], to demonstrate the generality of CIM-MLC. Experiments show that CIM-MLC can accelerate the inference speed of CIM work [27] and CIM work [29] by 2.3× and 3.7×, respectively, while reducing the peak power consumption by 75% for PUMA [4].

2 Background and Motivation

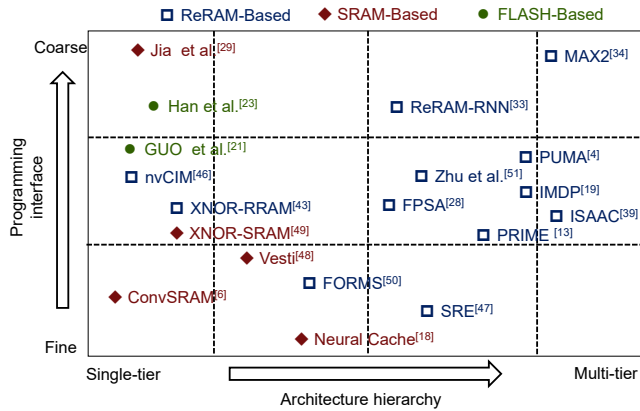### 2.1 The Diversity of CIM for DNNs

Figure 1. Diversity of CIM architecture.

Researchers have proposed various CIM-based DNN accelerators. We sort out the designs of recent CIM accelerators from three dimensions: memory device, architecture hierarchy, and programming interface, and summarize them in Figure 1 [4, 6, 13, 18, 19, 21, 23, 28, 29, 33, 34, 39, 43, 46–51]. SRAM, ReRAM, and FLASH are included in the dimension of the memory device. Their different read/write latency and storage density affect the data mapping and scheduling policy of these CIMs. For example, the write latency of ReRAM/FLASH is relatively long, so CIMs based on ReRAM/FLASH often ford write operations during computation (*e.g.*, Guo *et al.* [21] and PRIME [13]). As for the architecture hierarchy, some CIM designs hierarchically organize tiles, cores, and crossbars from top to bottom (*e.g.*,
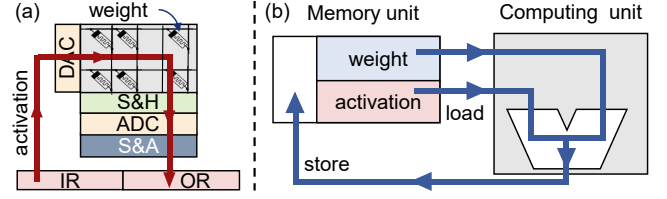

Figure 2. Computation dataflow comparison of (a) CIM and (b) traditional architecture.

ISSAC [39] and MAX2 [34]) while others do not (*e.g.*, a single-tier hierarchy only having tiled crossbars in ConvS-RAM [6] and nvCIM [46]). Additionally, even though ISAAC and PUMA [4, 39] have a similar hierarchical structure, they still use different crossbar sizes and the number of crossbars. Accordingly, their model mapping ways are different. The scheduling space during the compilation is different as well. The programming interface refers to the computing granularity that CIM can support from the users' view. As for the coarse-grained programming interface, the DNN computation is decomposed into convolution operations and then is performed by CIM cores (*e.g.*, Jia *et al.*'s work [29]), while for the fine-grained programming interface, the bit-wise vector operations are supported by CIM crossbars (*e.g.*, ConvSRAM [6]). Other works propose to decompose DNN operators and schedule MVM-grained operation in CIM crossbars [4, 19, 36, 43, 49]. To sum up, the diversity of existing CIMs for DNN raises a growing demand for an efficient general compilation tool.

2.2 Compilation Tools for CIM

Neural network compilers for different computing architectures have been widely discussed in prior works, such as TVM [9]. However, existing machine learning compilers primarily focus on compilation optimization related to traditional architecture hardware where the memory and computing units are separated [7, 10, 14, 20, 37, 38, 44]. As Figure 2 shows, traditional architecture and CIM are totally different computation paradigms. As for traditional architectures being bottlenecked by memory accesses, their compilation tools are composed of optimization methods for memory efficiency improvement. For instance, the popular loop unrolling and unfolding approaches in the traditional compilation tools are for improving data locality [31]. In contrast, CIM architectures performing computations inside memory [25, 39] shall focus on improving the efficiency of in-situ computation instead of improving the memory efficiency in the compilation techniques and hence desire a specific compilation tool.

Some researchers have noticed the problem and developed some CIM compilation tools [17, 22]. However, we observe these works are inadequate as a general compilation tool for CIM when summarizing them from the dimensions of

Table 1. Comparison of the generality of this work and the existing works.

	Supported Device Type			Supported Programming Interface			Optimization Granularity
	SRAM	ReRAM	MISC(*e.g.*, PCM, FLASH)	VVM	MVM	DNN Operators	/
PUMA [2, 4]	×	✓	/	×	✓	×	MVM
IMDP [19]	×	✓	/	✓	✓	×	MVM
TC-CIM [17]	×	✓	/	×	✓	×	MVM
Polyhedral-based [22]	×	✓	/	×	✓	✓	MVM, MM, Conv
OCC [40]	✓	✓	/	✓	✓	×	/
Ours	✓	✓	✓	✓	✓	✓	VVM, MVM, DNN Operators
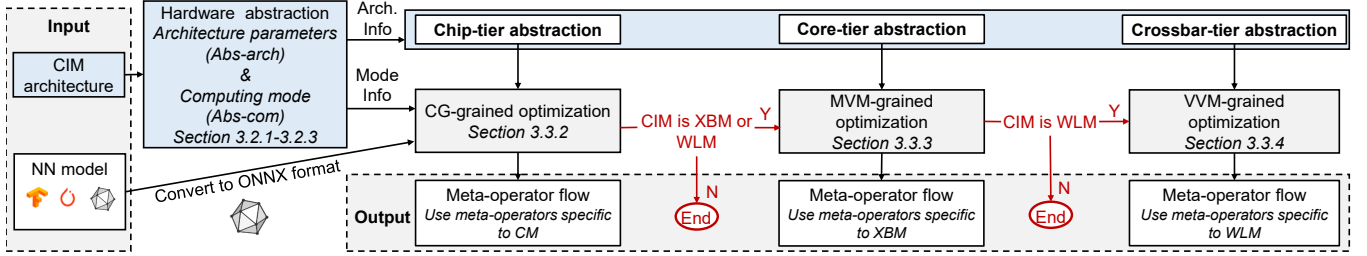
Figure 3. Overall workflow of CIM-MLC.

the device type, programming interface, and optimization granularity (as shown in Table 1).

Considering the device types involved, PUMA [2, 4] and IMDP [19] present compilation methodologies tailored to particular CIM architectures that rely on ReRAM. However, these approaches lack interoperability with alternative devices and programming interfaces. A few works have explored general compilation frameworks. TC-CIM [17] and Polyhedral-based [22] harness polyhedral models to facilitate the compilation process on CIM accelerators, which automatically identify and map MVM operations inherent in DNNs onto CIM. However, these two works also mainly focus on ReRAM-based CIM and assume that there are ample memory resources available for parameter loading or duplication, which overlooks crucial facts of resource-constrained situations that are typically encountered in SRAM-based CIMs. Relatively, OCC [40] is a comprehensive compilation that encompasses abundant device types as well as numerous programming interfaces. Built upon a specialized MILR with an ISA, this work enables the optimization at various levels of granularity. Nonetheless, this work does not incorporate the coarser-grained programming interface like the DNN operator and does not explore the mapping optimization of DNN computation on CIMs. To summarize, current research on CIM compilers lacks sufficient abstraction of hardware and is limited in support for various architectures and programming interfaces in CIM accelerators.

Regarding optimization granularity, most of the works [2, 4, 17, 22] support the mapping and optimization of DNN at the level of MVM operations. However, due to the increased complexity of the scheduling space, existing work has not

fully explored the optimization possibilities at the MVM granularity. For example, some of them [2, 4, 22] support the pipeline inter-network layers but do not consider the computing pipeline opportunity at the MVM-grained. In fact, optimizing the scheduling of MVM operations helps to further optimize computing efficiency.

3 Methodology

3.1 Overview

Figure 3 demonstrates the overview of the proposed CIM-MLC. Existing CIM compilation works have poor generality because of their single-level scheduling policy, while CIM-MLC is a general compiler that features unified abstraction from diverse hardware and multi-level scheduling with abundant meta-operators. The hardware abstraction provides the same description interface of architecture parameters (Abs-arch) and computing mode (Abs-com) to various CIM designs. To decouple the data mapping and computing scheduling with one architectural design, we propose the multi-level scheduling technology to handle the computing mode for different architectural tiers in CIMs. The multi-level scheduler tailors the optimization method for each computing mode, applies the optimization method independently or jointly according to the abstraction of the CIM accelerator, and finally generates the meta-operator flow for the CIM accelerator.

3.2 CIM Hardware Abstraction

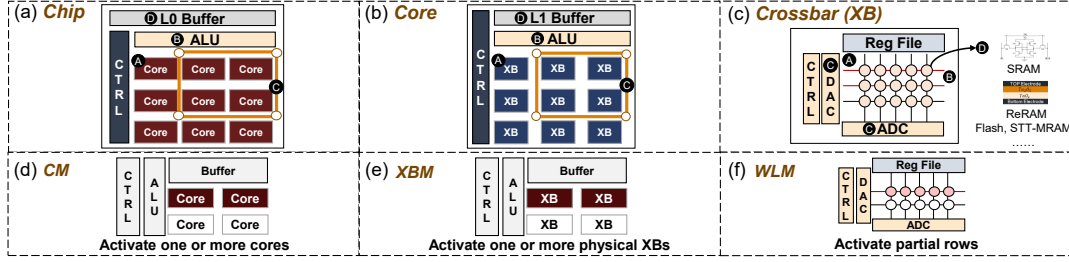In order to support a wide range of CIM accelerators, we first construct the hardware abstraction, covering two key

Figure 4. CIM Hardware Abstraction.

parameter	description	format
core_number	The number of cores in the chip	[number of cores per row * number of cores per column]
ALU	Digit computing capacity	X operations per second
core_noc	Network on chip(NoC) type	'Mesh', 'H-tree', etc
core_noc_cost	Network on chip(NoC) cost	matrix record the data transfer cost between each core
L0 size	Global buffer size	X kb
L0 BW	Global buffer bandwidth	X b/cycle

Figure 5. Chip tier architecture abstraction parameters.

parameter	description	format
xb_number	The number of xbs in the core	[number of xbs per row * number of xbs per column]
ALU	Digit computing capacity	X operations per second
xb_noc	NoC type	'Mesh', 'H-tree', etc
xb_noc_cost	NoC cost	matrix record the data transfer cost between each xb
L1 size	Local buffer size	X kb
L1 BW	Local buffer bandwidth	X b/cycle

Figure 6. Core tier architecture abstraction parameters.

aspects: the architecture parameters (Abs-arch) and the computing modes (Abs-com), respectively. We model the CIM-based DNN accelerator as a hierarchical architecture, which contains three tiers from top to bottom: (a) chip, (b) core, and (c) crossbar. Meanwhile, similar architectures may support different levels of CIM operation granularity. Some CIM designs only support specific operators or fixed-granularity MVM computation, while others offer interfaces for controlling rows, enabling a wide range of computation. So, we design a three-level computing abstraction for CIM. As illustrated in Figure 4(d)-(f), the top-to-bottom levels are Core Mode (CM), Crossbar Mode (XBM), and Wordline Mode (WLM), denoting the coarse-grained to fine-grained DNN operators, respectively. Architecture abstraction tiers and computing mode abstraction levels maintain a one-to-one correspondence. The hardware scheduling granularity provided by the CIM architecture determines the supported computing mode and the architecture abstraction parameters exposed to the compiler. We also develop a comprehensive set of meta-operators, which facilitate compilation scheduling optimization and instruction generation and enable users to define and customize hardware-supported operations within our framework.

3.2.1 Chip tier architecture abstraction and core mode.
As shown in the Figure 4(a), in the chip tier, multiple cores (Ⓐ) connected via a network-on-chip (NoC Ⓒ) are grouped into a single chip, each used for operators like convolution. A shared on-chip memory (Ⓓ) is used for data storage, and digital arithmetic logic units (ALU Ⓑ) are used for computations beyond the scope of CIM-supported operators.

At this tier, the minimum scheduling granularity provided by the CIM architecture is core. We abstract its computing mode as core mode (CM). In this mode, the compiler assigns one or more cores to complete one DNN operator (*e.g.*, convolution) according to the operator's demand and the core's capability. The scale of computation supported by a single core and the total number of cores determine the maximum number of operators that can be concurrently mapped on a chip. Hence, we employ the parameter **core_number** to record the total number of cores within the chip, thus abstracting the chip's CIM computational capacity.

The ALU performs commonly used neural network operators such as activation functions, pooling, and CIM-specific operations like shift-accumulate. ALU's functionality and computational speed impact the compilation of digital computation operators. We use **ALU** to denote the ALU computation speed and the functionality will be recorded by meta-operator, which we will show in Section 3.3.2.

The type of on-chip network and data transfer rate determine the data transfer efficiency between cores. We use **core_noc** and **core_noc_cost** to abstract the on-chip network. Meanwhile, as the storage capacity and bandwidth of the global buffer jointly determine the execution time of data movement, **L0 size** and **L0 BW** are employed to record on-chip buffer capacity and bandwidth.

The architecture parameters in chip tier are summarized in the Figure 5. We use the above mentioned parameters to abstract the CIM architecture characteristics in the chip tier and expose the parameters to the compilers in CM. CIM-MLC will combine the characteristics of operators in DNN and CIM architecture parameters to optimize latency and energy consumption during operator mapping.

3.2.2 Core tier architecture abstraction and crossbar mode.
As shown in the Figure 4 (b), the core tier abstraction includes the features within a core, which consists of multiple
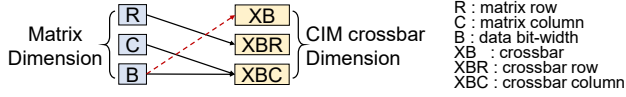
Figure 7. Dimension binding for DNNs mapping on CIM crossbar.

parameter	description	format
xb_size	The shape of crossbar	[number of cells per row * number of cells per column]
DAC	DAC precision	X bit
ADC	ADC precision	X bit
Type	type of storage cell	'SRAM', 'ReRAM', etc
Precision	precision of storage cell	X bit
parallel row	Maximum number of rows that can be activated simultaneously	

Figure 8. Crossbar tier architecture abstraction parameters.

crossbars (**A**), local buffers (**D**), and digital computation units (**B**). The crossbars are connected via NoC (**C**).

The minimum scheduling granularity at this tier is crossbar, which we abstract as crossbar mode(XBM). One or multiple crossbars work together to complete one matrix-vector multiplication in this mode. Similar to chip tier, we use the **xb_number** to record the number of crossbars that work at the same time, which can decide the maximum scale of matrix-vector multiplications that can be concurrently mapped on the core. The convolution operator in DNN is decomposed into a sequence of matrix-vector multiplications in the XBM.

To accommodate different CIM designs for executing MVM on crossbars [4, 39, 42, 51], we introduce the concept of *VXB (Virtual Crossbar)* as the computational unit rather than physical crossbars to facilitate the computing scheduling in the compiler. A dimension-binding scheme is designed to construct a VXB, specifying which crossbars collaborate to perform a single MVM. Figure 7 illustrates the dimension-binding scheme, where the matrix dimension has matrix row (R), column (C), and data bit-width (B). The computing crossbar dimension refers to the crossbar itself (XB), crossbar row (XBR), and column (XBC). Figure 7 shows that the matrix dimension R/C/B is bound to the XBR/XBC/XBC, respectively. This binding represents that the data bits are spread to the adjacent column in the crossbar. If the matrix dimension B is bound to the XB, the data bits will be spread to the different crossbars.

Similar to chip tier, we utilize **ALU** to denote the speed of the digital computation unit, **xb_noc** and **xb_noc_cost** for NoC abstraction, and **L1 size** and **L1 BW** to represent the local buffer characteristics at the core tier.

The architecture parameters in core tier are summarized in the Figure 6. These chip and core tier parameters are exposed to the compilers in XBM as core tier abstraction records the architecture details within the core, and chip tier parameters record the whole top tier architecture. CIM-MLC leverages

these parameters to map operators to crossbars, optimizing the crossbar utilization.

3.2.3 Crossbar tier architecture abstraction and wordline mode.
As shown in the Figure 4 (c), The crossbar tier abstraction serves as the fundamental computational unit, describing the details component within the crossbar. The crossbar tier has a memory crossbar (**A**) with its peripheral circuits (*e.g.*, wordline drivers, bitline drivers, and signal converters (ADC **C**, DAC **C**, etc.)). Each crossbar array's rows (**B**) can work independently.

At this tier, the minimum scheduling granularity provided by the CIM architecture is rows. We abstract this as wordline mode (WLM) and use **parallel row** to record the number of rows that can be activated concurrently. The mode allows the CIM to optimize the power cost or alleviate the variation by closing partial rows of one crossbar [27, 47]. Users can define the number of rows that can be activated simultaneously in a crossbar. Meanwhile, the shape of the crossbar determines the maximum scale of matrix-vector multiplication calculations. So, we use the parameter **xb_size** to record the size of a crossbar.

The attributes of memory cell (**D**) within the crossbar profoundly impact the computation process. Additionally, memory cell precision influences data representation and the requisite number of crossbars for operators, affecting scheduling decisions. Hence, we use the parameters **Type** and **Precision** to respectively record the memory cell type and precision.

For the peripheral circuits, the Analog-to-Digital Converter (ADC) or a special sense amplifier usually performs pre-defined calculations on the readout analog signals from the crossbar. The Digital-to-Analog Converter (DAC) is also needed to complete data signal conversion. The precision of DAC and ADC influences computation accuracy and latency. We use parameters **DAC** and **ADC** to record the precision of DAC and ADC.

The architecture parameters in crossbar tier are summarized in the Figure 8. We expose the whole three tiers of architecture abstraction parameters to the compiler in WLM. CIM-MLC will transform the computing process in DNNs into the row-wise read and write of the CIM crossbar.

3.3 Multi-Level Scheduling

With the architecture abstraction and computing mode, we apply a multi-level scheduling strategy to optimize the efficiency of DNN inference on CIM, achieving low-latency, high energy-efficient deployment of models. For each computing mode abstraction in the associated abstracted architecture tier, we design the optimization method.

Next, we introduce the whole workflow and the detailed optimization methods in the multi-level scheduling strategy.

3.3.1 Multi-Level Scheduling workflow.
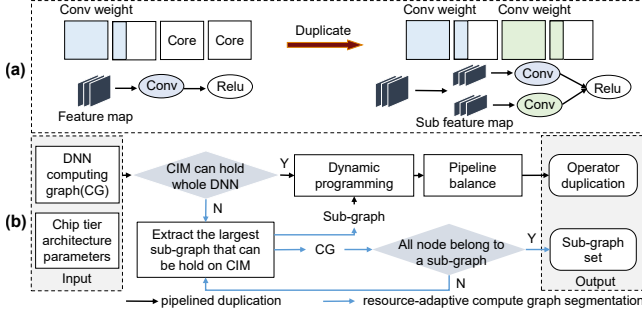The compilation process is illustrated in Figure 3. Firstly, the compiler

Figure 9. CG-grained optimization (a) Operator duplication illustration (b) Optimization algorithm.

gets the DNN models in ONNX format [5]. ONNX represents the model by computation graph, in which nodes correspond to operators, and edges denote the data dependency between each operator. The compiler progresses from coarse to fine granularity, optimizing the computations in Computational Graph Grained (CG-Grained), Matrix-Vector Multiplication Grained (MVM-Grained), and Vector-Vector Multiplication Grained (VVM-Grained). These optimizations are tailored to the CM, XBM, and WLM architectures. We adopt a multi-level joint scheduling optimization strategy to explore the scheduling space of the CIM architecture effectively.

When targeting the CM architecture, the compiler performs scheduling optimization at the CG-grained only. CG-grained optimization aims to explore operator duplication and pipeline strategies while considering resource constraints. The compiler takes the ONNX model and chip-tier hardware parameters as inputs. The optimization information for each operator is recorded by adding attributes to the nodes in the ONNX graph, such as the operator's duplication. For the XBM architecture, the compiler inherits the optimization results from the CG-grained, and chip and core-tier hardware parameters to perform the MVM-grained optimization. MVM-grained optimization further explores the finer operator duplication and pipeline before mapping operators to the crossbars. When dealing with the WLM mode architecture, the compiler builds upon the optimizations from the CG-grained and MVM-grained and, at a finer granularity of row scheduling, performs optimization at the VVM-grained.

3.3.2 CG-Grained Optimization. In the core mode, the compiler maps DNN operators onto cores following the chip tier architecture parameters. We propose a computing graph-grained optimization approach to improve operator mapping, all without altering the operator execution process. The primary objective of this optimization level is to judiciously utilize hardware resources, thereby reducing latency and power consumption.

Current graph-level optimization technology often overlook hardware characteristics [9]. We introduce a novel CG-grained optimization, exploring the *duplication* scheme and

Figure 10. The syntax of code generation in Backus Naur Form (BNF).

Figure 11. The syntax of CG-grained codegen in BNF format and MOP_CM semantics.

pipeline of DNN operators on CIM cores under the total `core_number` constraint. As shown in the Figure 9 (a), the operator duplication strategy enhances computational throughput by appropriately duplicating CIM-supported operators on cores. Meanwhile, the inter-operator pipeline strategy seeks to enhance execution efficiency.

Considering various hardware resource constraints of the CIM architecture, we propose a resource-adaptive compute graph segmentation and intra-segment dynamic balancing pipelined duplication algorithm as Figure 9 (b) illustrated. This algorithm takes the ONNX format computation graph and chip-tier hardware parameters as input and outputs the subgraph set of the computing graph and the duplication results for CIM-supported operators.

First, we initialize the resources and computation latency of each node. For convolution nodes, the computation latency scales with the output feature map size. We use dynamic programming to search for all operators' duplication numbers under the `core_number` constraint. Then, to avoid the pipeline stall because of the imbalance between computing and data access of adjacent layers, we adjust the duplication number for each node. Meanwhile, considering the constraint of `core_noc_cost` and `L0 BW`, CIM-MLC will update the duplication number to keep the data transfer amount within the NOC and buffer capability. Once the CIM-unsupported node, like Relu, follows the operator, we will also update the duplication number under the constraint of `ALU`.

As for the case that CIM resources are not able to hold a whole DNN, we iteratively construct the maximal sub-graphs that can fit within CIM capacity. For each constructed computation sub-graph, we update it by successively popping the last nodes from the subgraph. Then, we use the dynamic programming algorithm to get the latency of the remaining computation sub-graph after a node is popped out. Once the
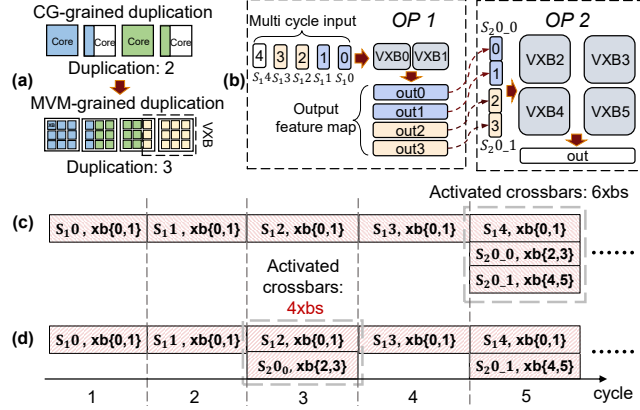
Figure 12. MVM-Grained Optimization (a) Operator duplication at VXB granularity (b) MVM-grained operator mapping and pipeline (c) Traditional pipeline activated crossbars (d) MVM-grained pipeline activated crossbars.

latency no longer decreases, the construction of that subgraph is complete. The nodes that pop out will be used to construct the next maximal sub-graphs until all nodes are part of a single sub-graph and have duplication numbers assigned.

Ultimately, we obtain the results of computing graph segmentation and operator duplication numbers at this grained optimization, which are subsequently passed on to the next optimization level.

Meta-operator Flow Generation After the optimization, the compiler backend binds operator execution to the corresponding cores. As primitives used in the current ML compiler [31] can not support the description of CIM, we introduce a CIM meta-operator set tailored for CM (MOP_CM) to describe the hardware activation at the chip tier. As the Figure 11 shows, MOP_CM includes the $CIM.read_{core}$ instruction. In addition to the CIM meta-operators, we have also designed corresponding meta-operators for other operations (*i.e.*, DCOM for digit computing operations and DMOV for data movement, respectively). Users have the flexibility to extend meta operators, aligning them with the hardware-supported functions. The compiler will compile relevant operators based on the provided meta-operators. The simplified syntax of code generation is shown in the Figure 10, in which the label parallel indicates the operators executing in parallel.

3.3.3 MVM-Grained Optimization. As for the XBM, the compiler unrolls the CIM-supported operator to matrix-vector multiply and then maps and schedules the MVM to crossbars. We design MVM-grained optimization after completing CG-grained scheduling, aiming to improve resource utilization and computing throughput during the slide of weight kernels on feature maps in the convolution by effectively using the crossbars in each core.

Specifically, MVM-grained optimization explores two key techniques: the *duplication* of the operator in the crossbars and MVM-grained computing *pipeline* to boost the computing throughput under the power limitation. The compiler receives the segmentation of the computing graph enriched with CG-level optimization information and chip and core-tier hardware abstraction, completing the MVM-grained optimization under the constraints of available `xb_number`.

We updated the *duplication* number of an operator within crossbars (as shown in Figure 12 (a)), which get the duplicated number using the following Equation:

$$D'^{O_i} = \lfloor \frac{num_{core}^{O_i} * D^{O_i} * Core_{VXB}}{num_{VXB}^{O_i}} \rfloor, \qquad (1)$$

where $Core_{VXB}$ is the number of VXBs in each core, D^{O_i} is the duplication number of the operator determined by the CG-grained optimization, $num_{core}^{O_i}$ is the number of cores occupied by this operator O_i, $num_{VXB}^{O_i}$ is the number of VXBs occupied by this operator O_i, which can be calculated based on the dimension and the size of VXB and the operator. Usually, one operator demands multiple VXBs to store its weights and complete the calculation.

MVM-grained computing *pipeline* strategically staggers the activation time of different crossbars to reduce peak power consumption as illustrated in Figure 12 (b). As we need to map a matrix-vector multiplication to multiple crossbars, we usually wait until all crossbars receive their inputs before computing in the traditional scheduling [39]. The pipeline strategy we propose, however, activates a crossbar as soon as it receives its input, completing the computing mapping to the crossbar. This reduces the number of crossbars that need to be activated simultaneously, thus lowering peak power consumption.

In the example, Operator 1 (OP 1) is mapped to VXB 0 and VXB 1, and Operator 2 (OP 2) is mapped to VXB 2-5. Input $S_1 i$, corresponding to one sliding window in convolution, enters OP 1's VXBs in sequence to perform the MVM of the convolution operator, passing its output to OP 2. The traditional approach waits four cycles for OP 2 to begin its computation with $S_2 0$.(Figure 12 (c).) In our MVM-grained pipeline, after OP 1 takes two cycles, OP 2 can start running on VXB 2,3 and then run on VXB4,5 after the next cycle, as shown in Figure 12 (d). At most, four VXBs are activated simultaneously in contrast to that 6 VXBs in Figure 12 (c), reducing the peak power by ∼30%. Meanwhile, OP 2's inputs, $S_2 0\_0$ and $S_2 0\_1$, are half the size of $S_2 0$ in the traditional pipeline. Thus, the communication overhead in each computing stage is reduced, alleviating the pressure on on-chip bandwidth and the risk of pipeline stall.

We obtain the updated duplication number and more compact pipeline in this optimization grained and we will pass the duplication result to the next optimization grained.

MVM-grained Codegen syntax	
<CIM> ::= <MOP_XBM>	
<MOP_XBM> ::= cim.read$_{xb}$(xb$_{addr}$,len)|cim.write$_{xb}$(xb$_{addr}$,mat)	
MOP_XBM Semantics	
cim.read$_{xb}$	The **len** xbs from the **xb$_{addr}$** are read, which complete the multiply of input and the data stored in the xbs
cim.write$_{xb}$	The **mat** is written to the **xb$_{addr}$**

Figure 13. The syntax of MVM-grained codegen in BNF format and MOP_XBM semantics.
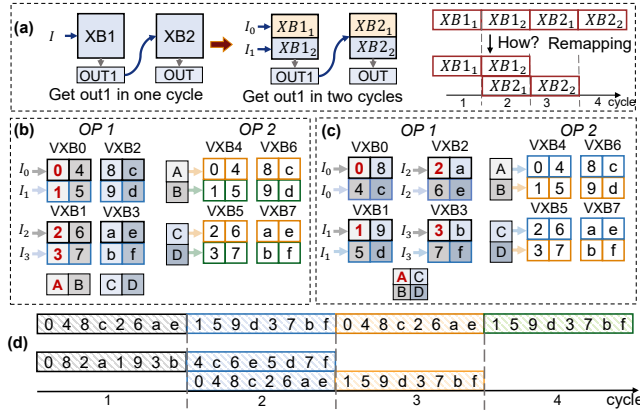

Figure 14. VVM-Grained Optimization (a) Parallelism opportunities in WLM (b) Naïve data mapping (c) Data remapping strategy (d) VVM-grained pipeline with naïve data mapping (up) and with our proposed data remapping (down).

Meta-operator Flow Generation Upon completing MVM-grained optimization, the compiler utilizes a meta-operator designed for XBM (MOP_XBM) to describe the hardware activation at the core tier. The syntax at this grained is shown in the Figure 10 and Figure 13. Specifically, MOP_XBM includes the $CIM.read_{crossbar}$ instruction for reading a specific crossbar to perform an MVM and $CIM.write_{crossbar}$ instruction for writing values like convolution weights to the crossbar.

3.3.4 VVM-Grained Optimization. In the WLM, partial rows within the crossbar can be activated at once, providing a more compact interface for vector-matrix multiplication compared to XBM which activates a whole crossbar for one computation. As shown in the Figure 14(a), when coarse-grained operators can be decomposed into fine-grained ones, we can explore additional parallelism opportunities. To further improve computing throughput, we propose an innovative data *remapping* strategy to enable a finer pipeline for the WLM CIM, which accounts for the updated computing graph with operator duplication results and whole three-tier abstraction.

The main idea of the remapping strategy is to distribute the data that contributes to the same computation to different crossbars. The proposed remapping strategy is illustrated in Figure 14, where OP 1 and OP 2 are two adjacent operators

VVM-grained Codegen Syntax	
<CIM> ::= <MOP_WLM>	
<MOP_WLM> ::= cim.read$_{row}$(row$_{addr}$,len)|cim.write$_{row}$(row$_{addr}$,value)	
MOP_WLM Semantics	
cim.read$_{row}$	The **len** rows from the **row$_{addr}$** are read, which complete the multiply of input and the data stored in the xbs
cim.write$_{row}$	The **value** is written to the **row$_{addr}$**

Figure 15. The syntax of VVM-grained codegen in BNF format and MOP_WLM semantics.

Table 2. Architecture Parameters of CIM example.

Chip_tier		Core_tier		Crossbar_tier	
core_number	[2*1]	**xb_number**	[2*1]	**xb_size**	[32*128]
				parallel row	16
				Precision	2-bit

in DNN and the output of OP 1 will be the input of OP 2. For the example in Figure 14 (b), **parallel row** is half of the **xb_size** rows, which means only row 0 and row 4 in VXB 0 can be activated in one cycle. In the naïve data mapping strategy (Figure 14 (b)), since OP 1's output A is the accumulation of output of rows 0, 1, 2, and 3, it needs two cycles to get the output A (Figure 14 (d) up). OP 2 can not start its computing until outputting A is complete. As a result, OP 2 has to delay its computation for one cycle. In contrast, our data remapping scheme maps rows 1 and 3 to different VXBs, as shown in Figure 14 (c). Then, rows 0-3 can complete their computations and accumulate to A in one cycle(Figure 14 (d) down). The OP 2 can start its computation at Cycle 2. Meanwhile, rows 4-7 can perform the computations and accumulate their results to get output B. Therefore, as shown in Figure 14 (d), a pipeline with a higher throughput can be achieved using our remapping scheme.

Meta-operator Flow Generation Upon completing VVM-grained optimization, the compiler uses the meta operators specific for WLM (MOP_WLM) to describe the corresponding hardware activation at the crossbar tier. As the Figure 15 shows, MOP_WLM includes the $CIM.read_{row}$ instruction to read rows and $CIM.write_{row}$ instruction to write certain values to the rows. CIM-MLC generates the meta-operator flow by invoking the MOP_WLM, DCOM, and DMOV, for which the syntax is shown in the Figure 10 and Figure 15.

3.4 Putting it all together

In this section, we put the CIM-MLC compilation process all together. To help enhance users' understanding of our hardware abstraction and optimization methods, we have employed simplified networks and the CIM architecture to illustrate our overall compilation process. We extracted commonly used operation in DNN, Convolution-Relu [1], as an example. The parameters of the convolution are: input
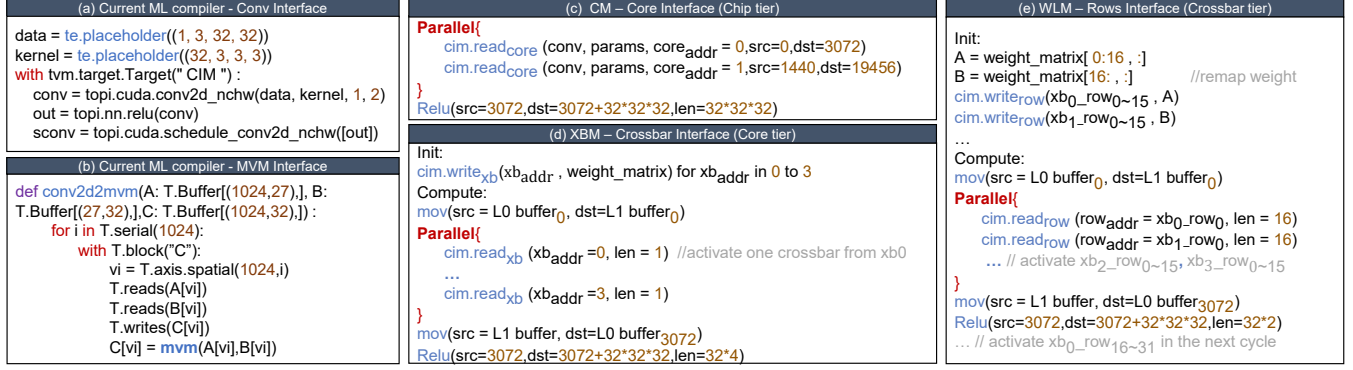
Figure 16. Generated code example for the Convolution-Relu. Left: Traditional DNN compilers; Right: CIM-MLC.

size:(3,32,32), kernel size:(32,3,3,3), stride:1, padding:1 with 8-bit precision for weight. We assume that the target CIM architecture has 2 cores, each housing 2 crossbars with 32 row × 128 column memory cells. Each cell is capable of mapping 2 bits. As we take a portion of the compilation process from the complete network as an example, we simplify this architecture to support all common digital arithmetic operations, and the buffer bandwidths are ample, imposing no memory access limitations on the computation process. We use shared memory communication as NoC example. The main architecture abstraction parameters are shown in the Table 2. We will illustrate the generated code when assuming the architecture provides activation interfaces for cores, crossbars, and rows, separately.

CG-Grained When this architecture only provides activation interfaces for cores, which means CM, we apply CG-grained optimizations.

In this stage, we duplicate operators based on the hardware resource constraint. In this example, **core_number** is 2 and each core can support the convolution with a kernel size of (32x3x3x3). Consequently, CIM-MLC decides the operator can be duplicated twice.

The final compiled meta-operator flow is illustrated in Figure 16(c) CM, which sequentially completes the convolution and ReLU operations. To enable the parallel calculation of the duplicated operators, we partitioned the input feature maps into several sub-ones with the same number as the duplicated operators. Then, we could get the buffer address of each sub-feature map. The duplication number for the operator is 2 and the feature map is partitioned into two sub-ones. So, we use two $cim.read_{core}$ to complete the convolution operation. As the two $cim.read_{core}$ have the same weights, their parameters are the same except for the src and des values. We use parallel{$cim.read_{core}$(conv,params,0,0,3072), $cim.read_{core}$(conv,params,1,1440,19456)} to denote executing computation on core 0 and core 1 in parallel. Furthermore, as we assume the architecture supports the ReLU operation, we can directly invoke the Relu meta-operator. So far, we have gotten the meta-operator flow for performing conv-relu operation on CM CIM by MOP-CM, DCOM and DMOV.

Traditional DNN compilers view this architecture merely as hardware for handling convolution and ReLU computations [31]. Therefore, during the CG level optimization, they may directly invoke these interfaces to perform computation or merge the operators to reduce memory access. The generated code may be like Figure 16 (a) [9]. Compared to our method, they do not account for CIM-specific characteristics such as duplication opportunities during the mapping phase, which leads to a restricted exploration of the optimization possibilities.

MVM-Grained When this architecture provides programming interfaces at the core tier, which means XBM, we can perform the corresponding optimization, MVM-Grained optimization. In this stage, we explore the operator duplication within a core after converting the convolution to MVM. Given that each core has two crossbars, our approach allows us to update the operator duplication from 2 to 4 as each crossbar can support an MVM.

As shown in the Figure 16(d) XBM, after determining the updated duplication number, we first write the weight matrix to corresponding crossbars by $cim.write_{xb}$ before computing. Then, we load the data into local buffers using mov operations and activate the four crossbars through the $cim.read_{xb}$ to complete four MVM operations in parallel. Since 1024 MVM operations are needed for one convolution, we generate 256 similar code segments to execute the entire convolution computation. Upon completing a batch of MVM operations, we perform ReLU calculations on the output of MVM to facilitate subsequent pipelines.

As we mentioned earlier in Section 1, traditional DNN compilation may be difficult to explore compilation optimizations suitable for CIM at MVM-grained. They usually focus on operators split and rearranged by loop unrolling *etc.* from the perspective of tensor size, but they cannot take advantage of the computing opportunity provided by the memory [31]. The generated code is similar to the one depicted in Figure 16 (b) [9]. For example, if we want TVM [9] to support the optimization at MVM-grained for CIM, we must first register plenty of templates for various CIM architectures. Then, we need to modify the whole optimization

pass of TVM, register the function interfaces supported by CIM, add new classes to expose the resources of the storage unit, and train a new automatic compilation optimization module. It is time-consuming and may completely destroy the existing compilation framework of TVM, which is not worth the loss.

VVM-Grained When this architecture offers rows activation interface within the crossbar, which means WLM, we will extend the data remapping at VVM-grained optimization, building upon the updated duplication.

Within the VVM optimization, we perform fine-grained control over the remapping of the weight matrix onto the crossbars. As the **xb_size** row is 32 and **parallel row** is 16, the original data that mapped to rows 0 to 15 (designated as 'A') and rows 16 to 31 (designated as 'B') in crossbar 0, is now divided into two distinct crossbars. 'A' is remapped to crossbar 0's rows 0 to 15, while 'B' is remapped to crossbar 1's rows 0 to 15 by the two $cim.write_{row}$ instructions. This data remapping enables the concurrent computation of 'A' and 'B' within a single cycle by $cim.read_{rows}$ in parallel, a marked improvement over the previous mapping where 'A' and 'B' required separate activation in two cycles to perform the cumulative calculation essential for a single MVM computation. This simultaneous calculation of 'A' and 'B' within a single cycle facilitates subsequent pipelines. Similar to code generation in the XBM, the compiler generated the meta-operators flow as shown in the Figure 16(e) WLM. A total of 512 similar compute blocks are needed to complete the convolution.

4 Experiment

This section presents the evaluation of CIM-MLC. We first present the implementation method of CIM-MLC, and then conduct a comprehensive comparison and evaluation of CIM-MLC with different CIM designs.

4.1 Experiment Setup

Simulator We developed a Python-based CIM functional simulator to verify the scheduling results, *i.e.*, the meta-operator execution trace. Additionally, we expanded upon an open-source simulator from previous studies [4, 8, 15] to evaluate the execution latency and power efficiency of the generated scheduling results.

We verify the effectiveness of the functional simulator by comparing it with the Pytorch framework [35]. In our built functional simulator, the hardware abstraction of CIM is described by a data structure, and meta-operators are implemented by specific functions. In this way, our functional simulator can perform the meta-operator flows as the DNN execution trace in the CIM.

We extended the open-source simulators proposed in previous works [4, 8, 15] as the performance simulator to support the execution cycle and power consumption evaluation

Table 3. Architecture Parameters of CIM Hardware Baseline.

Chip_tier		Core_tier		Crossbar_tier	
core_number	768	xb_number	16	xb_size	[128, 128]
ALU (ops/cycle)	1024	ALU (ops/cycle)	1024	parallel row	8
L0_BW	384 b	L1_BW	8192 b	DAC/ADC	1/8-bit
				Type/Precision	RRAM/2-bit

```
Computing_Mode= 'CM'
Chip_tier = {                Core_tier = {              XB_tier = {
  "core_number": 16           "xb_number": 1              "xb_size" :[1152,256]
  "ALU": \                    "ALU": \                    "parallel row":1152
  "core_noc":                 "xb_noc": \                 "DAC": 1-bit
  "Disjoint Buffer Switch"    "xb_noc_cost": \            "ADC": 8-bit
  "core_noc_cost": \          "L1 size": \                "Type": "SRAM"
  "L0 size": \                "L1 BW": \}                 "Precision" :1-bit}
  "L0 BW": \}
```

Figure 17. Architecture Abstraction of Jia *et al.*'s work [29].

```
Computing_Mode= 'XBM'
Chip_tier = {                Core_ tier = {             XB_ tier = {
  "core_number": 138          "xb_number": 2             "xb_size" :[128,128]
  "ALU": \                    "ALU": \                   "parallel row":128
  "core_noc": "mesh"          "xb_noc": \                "ADC": 1-bit
  "core_noc_cost": \          "xb_noc_cost": \           "DAC": 8-bit
  "L0 size": 96 KB            "L1 size": 1 KB            "Type": "ReRAM"
  "L0 BW": 384 b/cycle }      "L1 BW": \}                "Precision"  :2-bit}
```

Figure 18. Architecture Abstraction of PUMA [4].

```
Computing_Mode= 'WLM'
Chip_tier = {                Core_tier = {              XB_tier = {
  "core_number": 4            "xb_number": 2             "xb_size" :[256,64]
  "ALU": \                    "ALU": \                   "parallel row":32
  "core_noc ": \              "xb_noc": \                "DAC": 1-bit
  "core_noc_cost": \          "xb_noc_cost": \           "ADC": 6-bit
  "L0 size": \                "L1 size": \               "Type": "SRAM"
  "L0 BW": \}                 "L1 BW": \}                "Precision" :1-bit}
```

Figure 19. Architecture Abstraction of Jain *et al.*'s work [27].

of meta-operators flow on the CIM-based DNN accelerators. The primary extensions include: 1. We developed computational functions to facilitate simulating meta-operation execution, allowing the simulator to represent CIM architectures with varying computation granularity levels. 2. We establish the latency model, including computation, data movement, *etc.* to evaluate the overall DNN latency.

CIM Architecture Baseline We refer to ISAAC [39] to establish a CIM architecture and use it as the baseline. The parameters of the baseline architecture are listed in Table 3. The parameters that are not elaborated are considered ideal, indicating that their influence on the evaluation is disregarded. For example, if the on-chip buffer is assumed to have sufficient bandwidth, load/store time can be hidden within the computation time. We evaluate our CIM-MLC on the baseline to verify its effectiveness and compare it with Poly-Schedule [22], which is also a compilation work for CIMs.

Network Benchmark First, to verify the generality of the scheduling method for different neural network tasks, we tested our operator scheduling method on multiple classic network models, including the VGG series [41], ResNet series [24], visual transformer (ViT) [16], *etc.* All models'

weights and activation values are quantized with 8-bit precision and are tested on the ImageNet dataset.

Hardware Benchmark To verify the generality of CIM-MLC for different CIM architectures, we conduct experiments on three CIM-based accelerators [4, 27, 29] with different device types, precision, architecture hierarchy, and programming interfaces. Among them, Jia *et al.* [29] proposed an SRAM-based CIM accelerator that includes 16 CIMUs with a size of 1152x256, which incorporates embedded digital logic and high-precision ADC to achieve parallel activation calculation of 1152 rows. PUMA [4] is a programmable CIM architecture based on ReRAM to support a wide range of neural network applications. Jain *et al.* [27] introduces a CIM SRAM macro, in which only limited rows ($\le$ 32) can be activated simultaneously in a crossbar to alleviate the computing variation.

4.2 Effectiveness of CIM-MLC

In this section, we apply the CIM-MLC in existing CIM-based accelerators [4, 27, 29] to verify the generality of CIM-MLC and also compare CIM-MLC with previous work to show the effectiveness of our proposed optimization.

Firstly, we demonstrate the comparison results of CIM-MLC with these three different CIM-based accelerators. These works have not only the specific CIM architecture but also the performance optimization methods. So, we first verify our abstraction techniques on their CIM distinct architectures and then compare the optimized performance by our method with their performance.

Work 1: The hardware abstraction results of Jia *et al.*'s work are shown in Figure 17. The parameters that are not elaborated are considered ideal and denoted with "\". The computing mode abstraction of this design is the core mode (CM). CIM-MLC will apply CG-grained optimization to generate the meta-operator flow when mapping and scheduling the DNN. The performance comparison between CIM-MLC and Jia *et al.*'s work is shown in Figure 20 (a). The CG-grained P&D is the combination of the dynamic programming-based duplication and the pipeline, bringing about 3.7$\times$ speedup over Jia *et al.*'s work as it can make full use of the limited resources and speed up the computing of the bottleneck layer in DNNs. The pipeline strategy can only achieve the speedup by 1.2$\times$ over Jia *et al.*'s work because the model size exceeds on-chip resources and the performance would not be fully optimized without the data mapping design.

Work 2: Our hardware abstraction results for PUMA [4] are shown in Figure 18, where the computing mode is XBM. Thus, CIM-MLC can perform CG-grained and MVM-grained optimization. We compare PUMA with our work on the VGG16, and the results are shown in Figure 20 (b). When mapping the VGG16 model, the proposed MVM-grained optimization fully utilized the scheduling space of the XBM mode in PUMA architecture and introduced a fine-grained

MVM-grained pipeline. Our evaluation of peak power includes the power consumption of ADC/DAC, XB activation computation, and data movement. Our assessment shows that these three parts account for 10%, 83%, and 7%, respectively. Thus, CIM-MLC performs fine-grained time-division activation of XBs and associated ADC/DACs, thereby significantly reducing the peak power by 75%.

Work 3: Figure 19 presents the hardware abstraction of Jain *et al.*'s CIM macro [27] that has *WLM* computing mode. For a fair comparison, we use the VGG7 model as the benchmark and evaluate the scheduling results of CIM-MLC and the original result under the same resource constraints. As shown in Figure 20 (c), with the three-level scheduling optimization (*i.e.*, CG-grained, MVM-grained, and VVM-grained), we achieve a speedup of about 2.3$\times$ over Jain *et al.*'s work. Meanwhile, we show the speedup of CG-grained optimization and MVM-grained optimization, respectively. CG-grained optimization can achieve a speedup of 1.2$\times$, while MVM-grained optimization cannot further bring effective acceleration improvement. This is because this CIM macro has limited on-chip resources, especially the small number of XBs in a Core, leading to the ineffectiveness of MVM-grained optimization for improving the speedup. The VVM-grained optimization fully improves the computing pipeline efficiency between adjacent operators by converting serial computations into parallel computations, thereby achieving a speedup.

Comparison to CIM-oriented compilers: We compared our method with the existing general-purpose CIM compiler tool Poly-Schedule [22]. Poly-Schedule [22] supports the compilation of CIM-based accelerators for CM and XBM, using operator duplication techniques based on greedy strategies and batch pipeline strategies for acceleration. Compared with this work, CIM-MLC can optimize the internal computation pipeline of a single input image and explore the fine-grained scheduling space of the XBM mode to achieve better scheduling. We compared the operator scheduling results between Poly-Schedule and CIM-MLC in the same CIM architecture abstracted in Table 3. As shown in Figure 20 (d), compared to the latency result without any optimization, the Poly-Schedule [22] utilizes on-chip resources with a greedy strategy to reduce 84% computation cycles. Our work explores the fine-grained optimization space and reduces the computation cycles by up to 95%, which achieves about 3.2$\times$ speedup compared to the work [22].

4.3 Performance Analysis

In this section, we analyze the speedup results of different granularity scheduling optimization in the multi-level scheduling for the baseline architecture in Table 3. The network benchmark is the ResNet series [24]. The results are shown in Figure 21. In Figure 21 (a), we separate the optimization methods (*i.e.*, CG-Pipeline and CG-Duplication, CG-P&D) in the CG-grained optimization and investigate
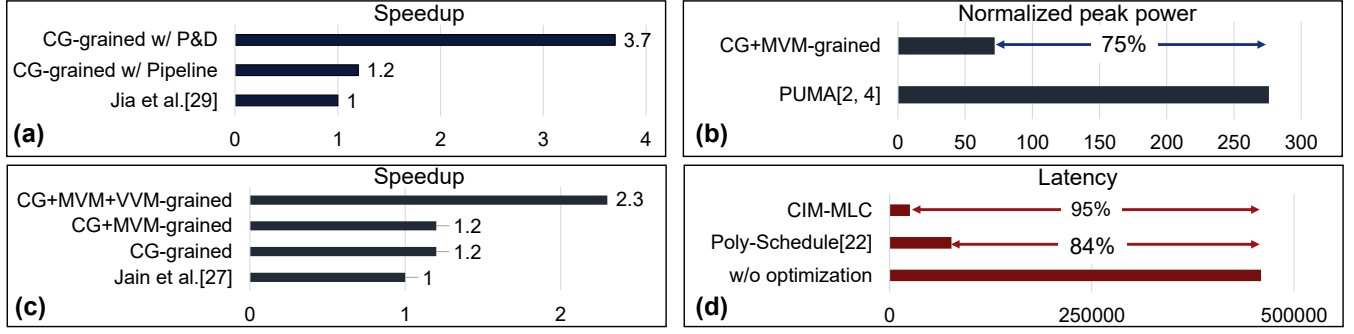
Figure 20. (a) Comparison of the speedup between this work and the schedule method in work [29]; (b) Comparison of the peak power consumption between this work and the schedule method in PUMA [2, 4]; (c) Comparison of the speedup between this work and the schedule method in work[27]; (d) Comparison of the latency between this work and the schedule method in work [22].
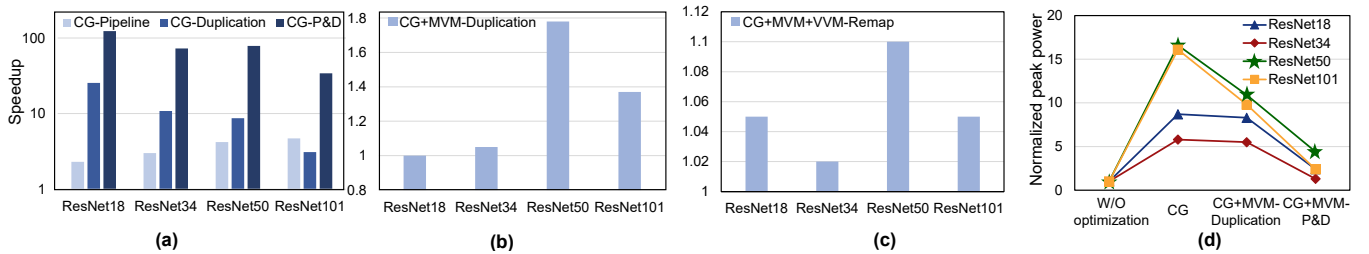

Figure 21. (a) Speedup of CG-grained optimization; (b) Speedup of CG+MVM-grained optimization; (c) Speedup of CG+MVM+VVM-grained optimization; (d) Comparison result of the peak power.

their results, and the results are normalized to the results of the baseline architecture in CM mode without any optimization. In Figure 21 (b)-(c), the fine-grained optimization is with the coarse-grained ones to show the advantage of the multi-grained optimization. The results of Figure 21 (b) are normalized to the CG-P&D results in Figure 21 (a), and Figure 21 (c)'s results use the results in Figure 21 (b) as the baseline.

It can be observed that the model structure has a significant impact on the scheduling results. As shown in Figure 21 (a), in CG-grained optimization, as the depth of ResNet increases, the speedup achieved by pipeline (CG-Pipeline) increases from 2.3× to 4.7×. The pipeline strategy can fully perform parallel computation between adjacent operators. However, with the increase of the model size (from ResNet18 to ResNet101), the speedup of duplication (CG-Duplication) decreases from 25.4× to 3.1×. When combining the pipeline and duplication strategies (CG-P&D), ResNet series models achieve significant performance improvements up to 123×.

As shown in Figure 21 (b), CG+MVM-Duplication increase the speedup of ResNet50/ResNet101 by approximately 1.8× / 1.4× over the CG-P&D. Meanwhile, as shown in Figure 21 (d), the normalized peak power consumption increases by approximately 5× to 16× for the ResNet series in CG-grained optimization since the number of crossbars that work at the

same time increases, leading to the high power consumption of CIM accelerator. Then, MVM-grained optimization reduces peak power consumption by up to 85% (ResNet101) thanks to its fine-grained pipeline strategy that lowers the peak activated crossbar number.

The results in Figure 21 (c) show that the speedup of ResNet50 in the VVM-grained optimization can be 10% higher than that of the MVM-grained optimization because the data remapping strategy improves the pipeline throughput.

4.4 Sensitive Study of CIM Architecture

In this section, we evaluate the effect of the change of CIM architecture parameters on the scheduling results of CIM-MLC. We use a transformer network architecture, ViT [16], as the benchmark. The baselined architecture uses the parameters from Table 3 except for the crossbar size is 128×256.

4.4.1 Core Number & Crossbar Number. First, we investigate the effect of different core numbers on the effectiveness of CIM-MLC. The results are shown in Figure 22 (a). As the total number of cores increases from 256 to 1024, the speedup achieved by CIM-MLC grows as well. Since on-chip resources gradually increase, CIM-MLC can fully show the potential of the duplication strategy as well as the pipeline. Therefore, the speedup of the CG-grained optimization increased from 15× to 30×. In the MVM-grained optimization,

Songyun Qu, Shixin Zhao, Bing Li, Yintao He, Xuyi Cai, Lei Zhang, and Ying Wang
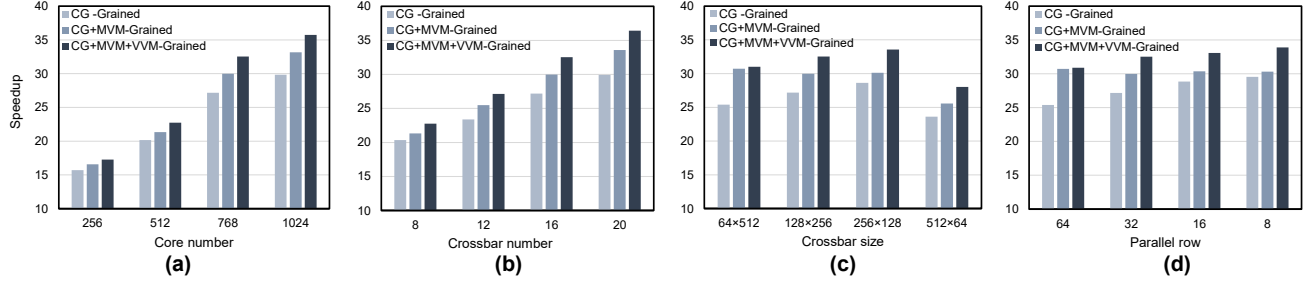

Figure 22. (a) Scheduling results with the different number of cores in a chip; (b) Scheduling results with different numbers of crossbars in a core; (c) Scheduling results with different crossbar sizes; (d) Scheduling results with different parallel rows in a crossbar.

as the number of cores increased, the proposed method could duplicate a single operator to different crossbars to fully utilize resources, achieving a maximum speedup improvement of about 1.1× compared to the CG-grained optimization. In the VVM-grained optimization, a finer pipeline reduced the equivalent activation times of the crossbar, further resulting in a speedup improvement of approximately 1.2× compared to the CG-grained result. Figure 22 (b) shows the performance speedup achieved by CIM-MLC when the number of crossbars in each core varies. Similar to the results of the core number, the speedup grows as the crossbar number increases.

4.4.2 Crossbar Size. Figure 22 (c) compares the speedup achieved by CIM-MLC as the crossbar size changes. As the crossbar row size increases from 64 to 256, the CG-grained speedup gradually increases. This is because larger row sizes increase the amount of input data required by the crossbar, which puts pressure on the bandwidth and leads to longer computing latency. The CG-grained pipeline optimization can reduce data feeding pressure, resulting in better acceleration. Since the weight matrix size of ViT matches the crossbar size when it changes from (64×512) to (256×128), the MVM-grained can make full use of redundant crossbar resources to achieve the same acceleration. The speedup of VVM-grained gradually increases because as the crossbar column size decreases, the same weight needs to be mapped on more crossbars in the horizontal direction. The VVM-grained remapping strategy can make full use of the advantages of these horizontal crossbars, which reduce the calculation delay in the crossbar, and achieve better acceleration. Upon increasing the number of crossbar rows to 512, the speedup has considerably decreased. This can be attributed to the fact that ViT comprises numerous matrices with a row size of 768, necessitating two vertical crossbars (512×64) for mapping. Consequently, ViT needs more resources, requiring it to be segmented and mapped on the given CIM, ultimately leading to a decrease in speedup.

4.4.3 Parallel Row. When the number of parallel rows in the crossbar is changed, the scheduling result can be observed in Figure 22 (d). If the number of parallel rows is decreased, it may be challenging to achieve effective acceleration through MVM-grained scheduling, while the VVM-grained remapping strategy can mitigate the impact of this reduction on latency. Specifically, when the number of parallel rows is 8, VVM-grained scheduling can achieve an acceleration of approximately 20% beyond the optimization results of MVM-grained scheduling.

5 Conclusion

We propose CIM-MLC, a general compilation tool for the CIM architecture that consists of hardware and computation abstraction and a multi-level operator scheduling method. The hardware abstraction method, Abs-arch, provides a unified description of CIM architectures from chip to crossbar tier. Three computing mode abstractions are established to represent different granularity of the programming interface. A set of meta-operators is established to describe the computing process under different modes. Besides, the multi-level scheduling optimization method is proposed to explore the acceleration potential of the CIM architecture, from CG-grained to MVM-grained and VVM-grained, and output the corresponding meta-operator flow. Comprehensive experimental results show that CIM-MLC has wide software and hardware adaptability. What's more, the proposed CIM-MLC can serve as the middleware between neural network models and CIM hardware, reducing the design burden on experts in the fields of CIM architecture and neural network algorithms.

6 Acknowledgments

References

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[2] Joao Ambrosi, Aayush Ankit, Rodrigo Antunes, Sai Rahul Chala-malasetti, Soumitra Chatterjee, Izzat El Hajj, Guilherme Fachini, Paolo Faraboschi, Martin Foltin, and Sitao Huang. Hardware-software co-design for an analog-digital accelerator for machine learning. In *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–13. IEEE, 2018.

[3] Shaahin Angizi, Zhezhi He, Dayane Reis, Xiaobo Sharon Hu, Wilman Tsai, Shy Jay Lin, and Deliang Fan. Accelerating deep neural networks in processing-in-memory platforms: Analog or digital approach? In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 197–202. IEEE, 2019.

[4] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, and Kaushik Roy. Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *Proceedings of the Twenty-Fourth International Conference on Archi-tectural Support for Programming Languages and Operating Systems*, pages 715–731, 2019.

[5] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[6] Avishek Biswas and Anantha P Chandrakasan. Conv-sram: An energy-efficient sram with in-memory dot-product computation for low-power convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 54(1):217–230, 2018.

[7] Xuyi Cai, Ying Wang, and Lei Zhang. Optimus: An operator fusion framework for deep neural networks. *ACM Transactions on Embedded Computing Systems*, 22(1):1–26, 2022.

[8] Pai-Yu Chen, Xiaochen Peng, and Shimeng Yu. Neurosim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(12):3067–3080, 2018.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, and Luis Ceze. Tvm: An automated end-to-end optimizing compiler for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[10] Weiwei Chen, Ying Wang, Ying Xu, Chengsi Gao, Cheng Liu, and Lei Zhang. A framework for neural network architecture and compile co-optimization. *ACM Transactions on Embedded Computing Systems*, 22(1):1–24, 2022.

[11] Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. A sur-vey of accelerator architectures for deep neural networks. *Engineering*, 6(3):264–274, 2020.

[12] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

[13] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Com-puter Architecture*, ISCA '16, pages 27–39, Piscataway, NJ, USA, 2016. IEEE Press.

[14] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christo-pher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel ngraph: An in-termediate representation, compiler, and executor for deep learning. 2018.

[15] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P Jouppi. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):994–1007, 2012.

[16] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weis-senborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, and Sylvain Gelly. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[17] Andi Drebes, Lorenzo Chelini, Oleksandr Zinenko, Albert Cohen, Henk Corporaal, Tobias Grosser, Kanishkan Vadivel, and Nicolas Vasi-lache. Tc-cim: Empowering tensor comprehensions for computing-in-memory. In *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*, 2020.

[18] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. Neu-ral cache: Bit-serial in-cache acceleration of deep neural networks. In *2018 ACM/IEEE 45Th annual international symposium on computer architecture (ISCA)*, pages 383–396. IEEE, 2018.

[19] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. In-memory data parallel processor. *ACM SIGPLAN Notices*, 53(2):1–14, 2018.

[20] Chengsi Gao, Ying Wang, Cheng Liu, Mengdi Wang, Weiwei Chen, Yinhe Han, and Lei Zhang. Layer-puzzle: Allocating and scheduling multi-task on multi-core npus by using layer heterogeneity. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[21] Xinjie Guo, F Merrikh Bayat, M Bavandpour, M Klachko, MR Mah-moodi, M Prezioso, KK Likharev, and DB Strukov. Fast, energy-efficient, robust, and reproducible mixed-signal neuromorphic classifier based on embedded nor flash memory technology. In *2017 IEEE International Electron Devices Meeting (IEDM)*, pages 6–5. IEEE, 2017.

[22] Jianhui Han, Xiang Fei, Zhaolin Li, and Youhui Zhang. Polyhedral-based compilation framework for in-memory neural network acceler-ators. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 18(1):1–23, 2021.

[23] Runze Han, Peng Huang, Yachen Xiang, Chen Liu, Zhen Dong, Zhiqiang Su, Yongbo Liu, Lu Liu, Xiaoyan Liu, and Jinfeng Kang. A novel convolution computing paradigm based on nor flash array with high computing speed and energy efficiency. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(5):1692–1703, 2019.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[25] Yintao He, Ying Wang, Cheng Liu, Huawei Li, and Xiaowei Li. Tare: task-adaptive in-situ reram computing for graph learning. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 577–582. IEEE, 2021.

[26] Yuquan He, Songyun Qu, Gangliang Lin, Cheng Liu, Lei Zhang, and Ying Wang. Processing-in-sram acceleration for ultra-low power visual 3d perception. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 295–300, 2022.

[27] Saurabh Jain, Longyang Lin, and Massimo Alioto. ±cim sram for signed in-memory broad-purpose computing from dsp to neural processing. *IEEE Journal of Solid-State Circuits*, 56(10):2981–2992, 2021.

[28] Yu Ji, Youyang Zhang, Xinfeng Xie, Shuangchen Li, Peiqi Wang, Xing Hu, Youhui Zhang, and Yuan Xie. Fpsa: A full system stack solution for reconfigurable reram-based nn accelerator architecture. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 733–747, 2019.

[29] Hongyang Jia, Murat Ozatay, Yinqi Tang, Hossein Valavi, Rakshit Pathak, Jinseok Lee, and Naveen Verma. 15.1 a programmable neural-network inference accelerator based on scalable in-memory comput-ing. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 236–238. IEEE, 2021.

[30] Bing Li, Bonan Yan, and Hai Li. An overview of in-memory processing with emerging non-volatile memory for data-intensive applications.

In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 381–386, 2019.

[31] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

[32] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[33] Yun Long, Taesik Na, and Saibal Mukhopadhyay. Reram-based processing-in-memory architecture for recurrent neural network acceleration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(12):2781–2794, 2018.

[34] Manqing Mao, Xiaochen Peng, Rui Liu, Jingtao Li, Shimeng Yu, and Chaitali Chakrabarti. Max 2: An reram-based neural network accelerator that maximizes data reuse and area utilization. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):398–410, 2019.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, and Luca Antiga. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[36] Songyun Qu, Bing Li, Shixin Zhao, Lei Zhang, and Ying Wang. A coordinated model pruning and mapping framework for rram-based dnn accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[37] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, et al. Glow: graph lowering compiler techniques for neural networks. corr abs/1805.00907 (2018). *arXiv preprint arXiv:1805.00907*, 2018.

[38] Amit Sabne. Xla: Compiling machine learning for peak performance. 2020.

[39] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 14–26, Piscataway, NJ, USA, 2016. IEEE Press.

[40] Adam Siemieniuk, Lorenzo Chelini, Asif Ali Khan, Jeronimo Castrillon, Andi Drebes, Henk Corporaal, Tobias Grosser, and Martin Kong. Occ: An automated end-to-end machine learning optimizing compiler for computing-in-memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(6):1674–1686, 2021.

[41] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[42] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE international symposium on high performance computer architecture (HPCA)*, pages 541–552. IEEE, 2017.

[43] Xiaoyu Sun, Shihui Yin, Xiaochen Peng, Rui Liu, Jae-sun Seo, and Shimeng Yu. Xnor-rram: A scalable and parallel resistive synaptic architecture for binary neural networks. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1423–1428. IEEE, 2018.

[44] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[45] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24,

1995.

[46] Cheng-Xin Xue, Je-Min Hung, Hui-Yao Kao, Yen-Hsiang Huang, Sheng-Po Huang, Fu-Chun Chang, Peng Chen, Ta-Wei Liu, Chuan-Jia Jhang, and Chin-I Su. 16.1 a 22nm 4mb 8b-precision reram computing-in-memory macro with 11.91 to 195.7 tops/w for tiny ai edge devices. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 245–247. IEEE, 2021.

[47] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I-Ching Tseng, Han-Wen Hu, Hung-Sheng Chang, and Hsiang-Pang Li. Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 236–249, 2019.

[48] Shihui Yin, Zhewei Jiang, Minkyu Kim, Tushar Gupta, Mingoo Seok, and Jae-Sun Seo. Vesti: Energy-efficient in-memory computing accelerator for deep neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(1):48–61, 2019.

[49] Shihui Yin, Zhewei Jiang, Jae-Sun Seo, and Mingoo Seok. Xnor-sram: In-memory computing sram macro for binary/ternary deep neural networks. *IEEE Journal of Solid-State Circuits*, 55(6):1733–1743, 2020.

[50] Geng Yuan, Payman Behnam, Zhengang Li, Ali Shafiee, Sheng Lin, Xiaolong Ma, Hang Liu, Xuehai Qian, Mahdi Nazm Bojnordi, and Yanzhi Wang. Forms: Fine-grained polarized reram-based in-situ computation for mixed-signal dnn accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 265–278. IEEE, 2021.

[51] Zhenhua Zhu, Hanbo Sun, Yujun Lin, Guohao Dai, Lixue Xia, Song Han, Yu Wang, and Huazhong Yang. A configurable multi-precision cnn computing framework based on single bit rram. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.