CIMFlow: An Integrated Framework for Systematic Design and Evaluation of Digital CIM Architectures

Yingjie Qi†, Jianlei Yang†§✉, Yiou Wang†, Yikun Wang†, Dayu Wang†, Ling Tang†,
Cenlin Duan‡, Xiaolin He†, Weisheng Zhao‡

† School of Computer Science and Engineering, Beihang University
‡ School of Integrated Circuit Science and Engineering, Beihang University
§ Qingdao Research Institute, Beihang University

Abstract—**Digital Compute-in-Memory (CIM) architectures have shown great promise in Deep Neural Network (DNN) acceleration by effectively addressing the "memory wall" bottleneck. However, the development and optimization of digital CIM accelerators are hindered by the lack of comprehensive tools that encompass both software and hardware design spaces. Moreover, existing design and evaluation frameworks often lack support for the capacity constraints inherent in digital CIM architectures. In this paper, we present CIMFlow, an integrated framework that provides an out-of-the-box workflow for implementing and evaluating DNN workloads on digital CIM architectures. CIMFlow bridges the compilation and simulation infrastructures with a flexible instruction set architecture (ISA) design, and addresses the constraints of digital CIM through advanced partitioning and parallelism strategies in the compilation flow. Our evaluation demonstrates that CIMFlow enables systematic prototyping and optimization of digital CIM architectures across diverse configurations, providing researchers and designers with an accessible platform for extensive design space exploration.**

Index Terms—**Digital Compute-in-Memory, Integrated Framework, Instruction Set Architecture, Compilation**

I. INTRODUCTION

Recent advances in Deep Neural Networks (DNNs) have led to unprecedented achievements across various domains, significantly increasing the demand for efficient processing of deep learning workloads. However, traditional von Neumann architectures [1], [2] are hitting the "memory wall" due to frequent data transmission between separate computing and memory units [3]. As AI technologies continue to transform computing, Compute-in-Memory (CIM) architectures [4]–[6] have emerged as a promising solution for next-generation DNN accelerators, aiming to eliminate this bottleneck by integrating computation logic within memory arrays.

Mainstream CIM designs can be broadly categorized into analog [7]–[9] and digital [10]–[12] approaches. While analog CIM relies on current/voltage summation for computations, digital CIM embeds digital logic units directly into SRAM arrays. By avoiding analog-to-digital (ADC) and digital-to-analog (DAC) conversion overhead and non-ideality issues inherent in analog computations, digital CIM demonstrates both

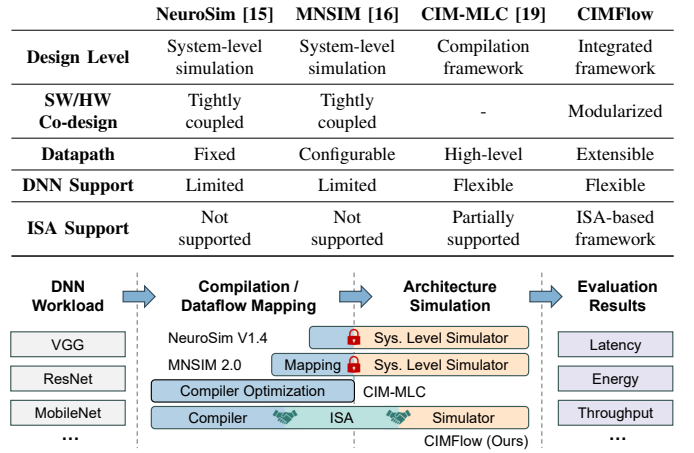|	NeuroSim [15]	MNSIM [16]	CIM-MLC [19]	CIMFlow
Design Level	System-level simulation	System-level simulation	Compilation framework	Integrated framework
SW/HW Co-design	Tightly coupled	Tightly coupled	-	Modularized
Datapath	Fixed	Configurable	High-level	Extensible
DNN Support	Limited	Limited	Flexible	Flexible
ISA Support	Not supported	Not supported	Partially supported	ISA-based framework

Fig. 1. Comparing CIMFlow with recent design and evaluation frameworks for CIM architectures.

robust computation and enhanced parallelism, thus exhibiting great potential in DNN acceleration.

However, fully realizing the benefits of digital CIM across various DNN workloads is fraught with challenges. Current CIM architectures, like many domain-specific accelerators, are typically optimized for specific applications, limiting their adaptability to emerging workloads. As the field of AI continues to evolve, designers face a constantly expanding design space, comprising diverse DNN structures, dataflow mapping strategies, and various design choices in CIM accelerators. Navigating this complex landscape requires versatile tools capable of accommodating a wide range of software and hardware configurations.

While various software tools have been developed to facilitate CIM designs, existing approaches have notable limitations. On the one hand, many tools tend to focus predominantly on specific aspects of the design flow, such as hardware simulation [13]–[16] or dataflow compilation [17]–[19], lacking the holistic view necessary for effective design space exploration. On the other hand, most of these tools are primarily designed for analog CIM, and are only later adapted to support digital implementations, often overlooking crucial characteristics of digital CIM architectures.

In particular, SRAM-based digital CIM faces inherent capacity constraints due to its lower integration density compared to DRAM or emerging NVM solutions [20]. In practical

implementations, these density limitations often result in insufficient on-chip memory capacity for modern DNN models. This constraint could potentially negate the benefits of in-memory computing, as it necessitates frequent data movement and restricts opportunities for inter-layer pipeline optimization. Consequently, there is a pressing need for an integrated framework specifically tailored to the nuances of digital CIM, enabling comprehensive evaluation and rapid prototyping across diverse configurations.

To address these challenges, we introduce CIMFlow, an integrated framework for systematic design and evaluation of digital CIM architectures. As shown in Fig. 1, our framework integrates a highly extensible Instruction Set Architecture (ISA), a digital CIM-oriented compiler, and an efficient cycle-accurate simulator, enabling comprehensive and flexible exploration of diverse software and hardware configurations. The main contributions can be summarized as follows:

- We propose CIMFlow, an integrated framework that provides an out-of-the-box workflow for implementing and evaluating DNN workloads on digital CIM architectures. The framework enables systematic design space exploration through seamless integration of compilation and simulation infrastructures.
- We develop a CIM-specific ISA design that employs a hierarchical hardware abstraction, bridging compilation and simulation while providing flexible support for various architectural configurations.
- We implement an advanced compilation flow built on the MLIR infrastructure, addressing the capacity limitations in digital CIM architectures through innovative partitioning and parallelism strategies.

II. BACKGROUND AND MOTIVATION

A. Digital CIM Preliminaries

CIM architectures represent a paradigm shift from traditional von Neumann computing by enabling computation directly within memory arrays. In essence, CIM architectures perform matrix-vector multiplication (MVM) through parallelized in-situ Multiply-Accumulate (MAC) operations within memory arrays, significantly reducing the data movement bottleneck that plagues conventional architectures. These MAC operations are typically carried out in a bit-serial fashion, where multiplications are decomposed into a series of bit-wise computations, followed by shift and accumulation operations.

A digital CIM macro comprises two key components: (1) a modified SRAM array that stores weight data and enables in-situ computations, and (2) peripheral circuits that orchestrate row-wise Boolean operations and accumulation [21]. Unlike analog CIM approaches, which face significant limitations due to area and power overhead from ADC and DAC converters, digital CIM enables simultaneous activation of the entire memory array. This enhanced parallelism, combined with the inherent robustness of digital computation, makes digital CIM particularly well-suited for accelerating deep learning workloads that exhibit high computational intensity and parallel structure. While researchers have proposed various digital CIM accelerators to harness these benefits [22], [23], scaling these solutions to meet the diverse computational demands of modern AI models remains a significant challenge.

B. Related Works and Our Motivation

The growing complexity of deep learning workloads and CIM architectures has created a critical need for tools and frameworks to bridge the gap between hardware and software. Existing frameworks broadly fall into two categories: modeling/simulation frameworks that evaluate architectural decisions through detailed performance analysis [13]–[16], and compilation frameworks that enable efficient mapping of DNN workloads to CIM hardware [17]–[19]. While modeling frameworks help hardware designers optimize latency, power, and area trade-offs, compilation frameworks provide support for managing the hardware-software interface and generating optimized dataflows for CIM architectures.

However, most existing frameworks have focused primarily on analog CIM architectures, given their prominence in current research. As illustrated in Fig. 1, recent simulation frameworks have begun extending support towards digital CIM implementations. NeuroSim V1.4 [15] extends its system-level simulation capabilities to support advanced digital CIM technology nodes, while MNSIM 2.0 [16] proposes a unified memory array model for both analog and digital CIM architectures. Although these tools provide valuable insights into architectural trade-offs, they often employ tightly-coupled co-design with fixed assumptions about DNN structures and datapath organizations, limiting their flexibility and applicability for comprehensive digital CIM exploration. On the compilation front, while traditional deep learning compilation infrastructures offer general-purpose solutions [24], frameworks such as CIM-MLC [19] introduce optimization strategies specifically designed for analog CIM. However, there remains a notable lack of dedicated compilation support for digital CIM architectures.

The above observations highlight three key challenges in digital CIM development. ❶ **Integration Gap:** The lack of comprehensive integration between different design stages necessitates an integrated framework that combines ISA definition, compilation, and simulation. ❷ **Limited Flexibility:** Fixed architectural assumptions in current tools might constrain design space exploration, calling for extensibility in CIM design and evaluation frameworks. ❸ **Resource Constraints:** The limited SRAM array capacity in digital CIM requires sophisticated dataflow management and parallelism strategies that existing works do not adequately address. To tackle these challenges, we propose CIMFlow, an integrated framework that enables flexible design space exploration through extensible ISA support and advanced compilation optimization techniques. By providing comprehensive support across the design stack, CIMFlow facilitates structured development and evaluation of digital CIM architectures while maintaining the flexibility to accommodate future innovations.

III. CIMFLOW FRAMEWORK

A. Framework Overview

The design of CIMFlow is guided by two core principles: **integration** for seamless exploration, and **extensibility** through
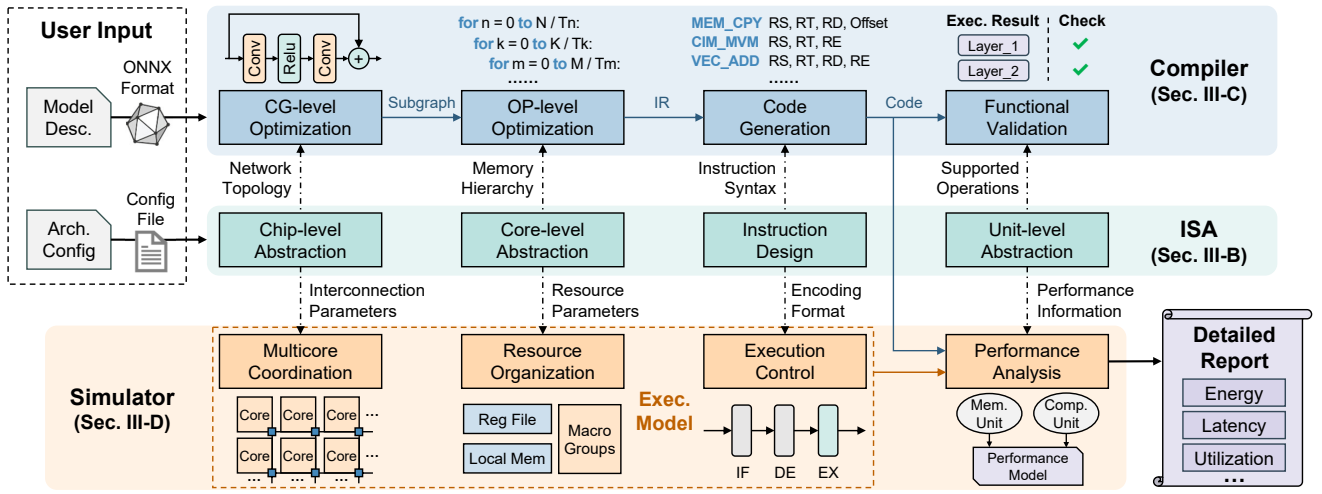
Fig. 2. Overview of the CIMFlow framework.

hierarchical abstractions. CIMFlow unifies the entire workflow from high-level DNN model description to detailed performance analysis, providing researchers with a comprehensive view of DNN workload execution on digital CIM architectures. Built with a modular design, CIMFlow readily adapts to emerging advances in CIM technology while abstracting implementation complexities through its streamlined interface, enabling designers and researchers to perform systematic evaluation and exploration.

As illustrated in Fig. 2, CIMFlow offers a cohesive workflow through three main components: a flexible ISA with hierarchical hardware abstraction (Sec. III-B), a compiler featuring multilevel optimization techniques (Sec. III-C), and a cycle-accurate simulator delivering detailed performance insights (Sec. III-D). The workflow begins with a DNN model description in ONNX format, complemented by an architecture configuration file that specifies the target CIM hardware parameters. The compiler first performs computational graph (CG) level optimizations for workload distribution and data management, followed by operator (OP) level optimizations to further maximize hardware efficiency. These optimizations are guided by the hardware specifications defined through the hierarchical hardware abstraction interface in ISA, spanning from chip-level interconnection to unit-level execution details. The generated code is validated by the compiler and then fed into the cycle-accurate simulator, which models the execution across multiple cores while tracking resource utilization and performance metrics. This process produces a detailed report covering energy consumption, latency, and hardware utilization, enabling comprehensive evaluation of different architecture designs.

B. ISA Design

As depicted in Fig. 3, the CIMFlow ISA implements a three-level hardware abstraction hierarchy complemented by a flexible instruction set design. Each abstraction level interfaces with the corresponding stages in the compilation and simulation infrastructure, providing architectural specifications that guide both compilation optimization and simulation execution.
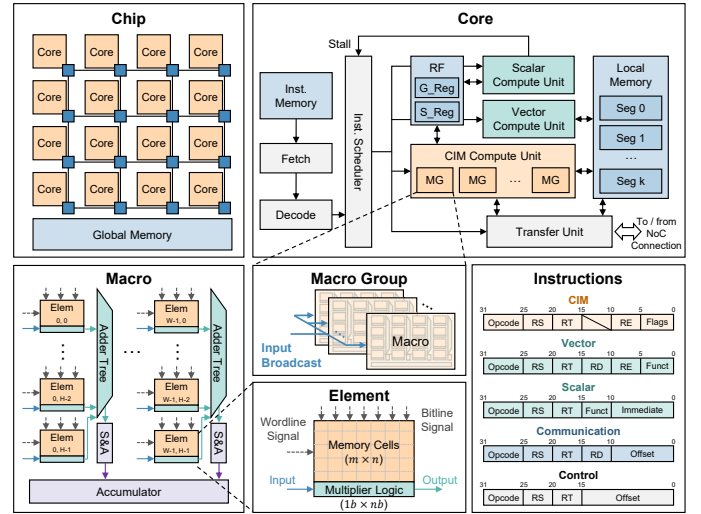

Fig. 3. Hardware abstraction and instruction design in the CIMFlow ISA.

Hardware Abstraction. At chip level, the architecture consists of multiple cores interconnected through a Network-on-Chip (NoC) structure, facilitating synchronous inter-core communication and global memory access. This organization enables scalable workload distribution and flexible inter-core pipelining, with each core functioning as a basic unit of program execution with its own instruction control flow.

The core-level abstraction defines the organization of hardware resources, encompassing instruction memory, various compute units, register files (RFs), and local memory. To facilitate efficient memory management and architectural extensibility, CIMFlow implements a unified address space across both global and local memories. In addition, the local memory is divided into segments to efficiently handle the input and output of DNN layers. The register file consists of general-purpose registers (G_Reg) for instruction-level access and special-purpose registers (S_Reg) for operation-specific functions.

At unit level, the CIM compute unit incorporates multiple macro groups (MGs) that support weight duplication and
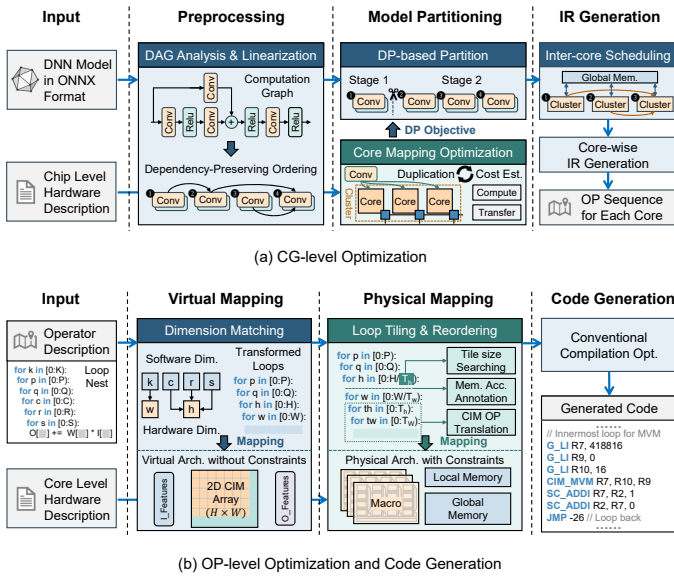
(a) CG-level Optimization

(b) OP-level Optimization and Code Generation

Fig. 4. Compilation flow and mapping optimization strategies in CIMFlow.

Algorithm 1: DP-based partitioning and mapping

Input: Preprocessed computation graph $G = (V, E)$,
Hardware resources R

Output: Optimal partitioning and mapping solution S

1   $D \leftarrow$ GetDependencyMasks(G)   // Find all dependency closures in G and encode them as bitmasks

2   $dp \leftarrow [\infty]^{|D|}, prev \leftarrow [-1]^{|D|}, map \leftarrow [\emptyset]^{|D|}$

3   **for** $i \leftarrow 0$ **to** $|D| - 1$ **do**

4   **if** $D[i] = \emptyset$ **then**

5         $dp[i] \leftarrow 0$

6   **continue**

7   **for** $j \leftarrow 0$ **to** $i - 1$ **do**

8   **if** $D[i] \& D[j] = D[j]$ **then**

9            $stage \leftarrow D[i] - D[j]$ // Extract the set difference of dependencies as a partition

10            $(cost, mp) \leftarrow$ OptimalMapping($stage, R$)

11   **if** $dp[j] + cost < dp[i]$ **then**

12               $dp[i] \leftarrow dp[j] + cost$

13               $prev[i] \leftarrow j$

14               $map[i] \leftarrow map[j] \cup mp$

15   $S \leftarrow$ ReconstructSolution($dp, prev, map$)

16   **return** S

---

flexible spatial mapping strategies. Within each MG, weights are typically organized along the output channel, enabling efficient input data broadcast across macros for parallel in-memory MVM operations. The vector compute unit handles auxiliary DNN operations such as activation, pooling, and quantization. The scalar compute unit executes control flow operations through scalar arithmetic computations.

Instruction Design. To support efficient execution across the hardware hierarchy, CIMFlow implements a unified 32-bit instruction format with specialized variations for different operation types. Instructions are categorized into compute, communication, and control flow instructions, with compute instructions further specialized for CIM, vector, and scalar compute units. Each instruction contains a 6-bit operation specifier (opcode) and multiple 5-bit operand fields. Certain instruction types may also include supplementary fields, such as a 6-bit functionality specifier, execution flags, or immediate values of 10 or 16 bits. The instruction format supports up to four operands depending on the operation type, providing flexibility for complex operations while maintaining encoding efficiency. The instruction set is designed for extensibility through incorporating a customized instruction description template, which enables seamless integration of new operations into the framework when provided with their associated performance parameters.

C. Compilation Flow

The CIMFlow compiler bridges the semantic gap between high-level DNN models and low-level CIM operations through a two-level optimization strategy, as illustrated in Fig. 4. Starting with an ONNX model, the compiler first performs CG-level optimizations to partition and schedule workloads across multiple cores, effectively addressing the limited capacity issue. This is followed by OP-level optimizations built upon the MLIR infrastructure [24], which translates DNN operations

into efficient CIM instruction sequences while taking into account the underlying hardware constraints.

CG-level Optimization. The optimization at this level begins with preprocessing the computation graph through analyzing the operator dependencies within the directed acyclic graph (DAG). During preprocessing, the compiler first identifies and extracts MVM-based operators, then groups adjacent operators with them to create a condensed CG. This analysis produces a dependency-preserving linear sequence of operators that forms the foundation for subsequent optimization stages.

To address the capacity limitation inherent in digital CIM architectures, CIMFlow implements a systematic partitioning strategy to divide the model into multiple execution stages. As detailed in Alg. 1, the model partitioning phase employs a dynamic programming (DP) based approach that optimizes workload distribution across available cores. The algorithm incorporates a state compression optimization that encodes all the dependency closures in the DAG as bitmasks, significantly reducing both space complexity and computational overhead. Each dependency closure represents a self-contained set of operators whose dependencies are fully enclosed within the set, serving as basic building blocks for candidate partitions.

The compiler derives candidate partitions through set operations on these dependency closures, and performs core mapping optimization for each partition. This process involves strategically duplicating operator weights across clusters of cores when deemed beneficial by the cost estimation model. To balance parallel execution benefits against communication costs, the estimation model accounts for both computation costs and data transfer overheads across inter- and intra-cluster communications. These cost assessments and their corresponding optimal mapping configurations are then used to guide the

DP-based partition selection.

The final phase of CG-level optimization focuses on inter-core scheduling and intermediate representation (IR) generation. The scheduler orchestrates data movement through the NoC interconnection, facilitating the inter-operator pipelines across different clusters. For each core, the compiler generates an optimized operation sequence incorporating both partitioning decisions and mapping destinations, establishing the foundation for OP-level optimizations.

OP-level Optimization. Following CG-level workload distribution, the compiler performs fine-grained operator transformations to maximize hardware efficiency. This process involves a structured approach that first establishes an ideal mapping in a constraint-free virtual space, and then adapts this mapping to actual hardware resource constraints.

The virtual mapping phase begins by analyzing the dimensional structure of each operator, transforming complex nested loops into a simplified version that aligns with the CIM array structure. This transformation process maps the software-level weight dimensions onto a two-dimensional array representation. By temporarily abstracting away physical constraints, the compiler explores the optimal weight data layout strategies, including the image-to-column (im2col) transformation commonly employed in DNN acceleration.

The physical mapping phase then adapts the idealized representation to actual hardware constraints through a series of optimization passes implemented within the MLIR infrastructure. The compiler first applies loop tiling based on resource capacity constraints, then systematically extracts MVM operations from the tiled loops for translation into CIM operations. Through automated analysis, it determines the optimal tile sizes and loop ordering to maximize computational efficiency while respecting resource limitations at each memory hierarchy. Memory access operations are then strategically annotated at appropriate loop levels to minimize data transfer overhead.

In the final code generation phase, the optimized IR undergoes conventional compilation techniques, including constant propagation, dead code elimination, and register allocation. The generated instructions adhere to the CIMFlow ISA specification while realizing the optimized resource mapping decisions, ensuring efficient utilization of the CIM hardware resources.

D. Simulator Design

The CIMFlow simulator provides cycle-accurate performance analysis through detailed modeling of the digital CIM architecture across multiple abstraction levels, from individual core execution to chip-level coordination. Implemented in SystemC [25], the simulator features a detailed pipeline model to track execution flow and resource utilization within each processing unit, while managing parallel execution across cores connected via NoC. The simulator supports diverse architectural configurations through a user-defined configuration file that adheres to the ISA specifications, while its modular design and standardized interfaces allow straightforward integration of custom architectural components.

TABLE I
ARCHITECTURE PARAMETERS OF THE DEFAULT ARCHITECTURE.

Chip Level		Core Level		Unit Level	
Core num.	64	CIM comp. unit	16 # MG	Macro	512×64
NoC flit size	8 Byte	Macro group	8 # macro	Element	32×8
Global mem.	16 MB	Local mem.	512 KB		

At core level, instruction execution follows a three-stage pipeline comprising instruction fetch (IF), decode (DE), and execute (EX). The EX stage implements detailed execution models for different compute units, each with fine-grained pipelining to enable instruction-level parallelism. Instruction conflicts and resource utilization are efficiently tracked through a bitmap-based scoring board within the instruction scheduler, ensuring accurate modeling of both computation and data movement patterns. Through this detailed modeling of both computation and data movement patterns, the simulator provides a comprehensive performance analysis across different architectural levels, tracking metrics such as energy consumption, execution latency, and hardware utilization for each unit. These detailed insights enable both quantitative evaluation of different CIM design choices and validation of compiler optimizations.

IV. EXPERIMENTAL RESULTS

A. Experimental Setup

To demonstrate how CIMFlow facilitates digital CIM architecture design, we conduct detailed analyses of compilation optimization strategies and present case studies exploring the impact of various architectural and software design choices. The default architecture parameters are carefully selected to efficiently support typical DNN workload characteristics while maintaining practical hardware constraints, as detailed in Tab. I. The performance statistics are acquired from multiple industry-standard tools to ensure accurate modeling of all components within the architecture. The CIM macro specifications are derived from post-layout analysis based on the design presented in [11], while other on-chip memory components are evaluated using memory compilers. The remaining digital logic modules are implemented in Verilog HDL and synthesized using Design Compiler, with power analysis conducted through PrimeTime PX. The NoC interconnection costs are modeled using Noxim [26].

We select representative DNN models that span different architectural characteristics and computational demands as our evaluation benchmark. The suite encompasses compute-intensive architectures including ResNet18 and VGG19, alongside compact models featuring depth-wise separable convolutions such as MobileNetV2 and EfficientNetB0. To align with digital CIM implementation constraints, the weights and activations of all models are quantized to INT8.

B. Compilation Optimization Evaluation

We evaluate our proposed compilation optimization strategies against two baseline approaches: (1) a generic mapping scheme that implements inter-layer pipeline without operator duplication, and (2) the CG-level partition and opportunistic
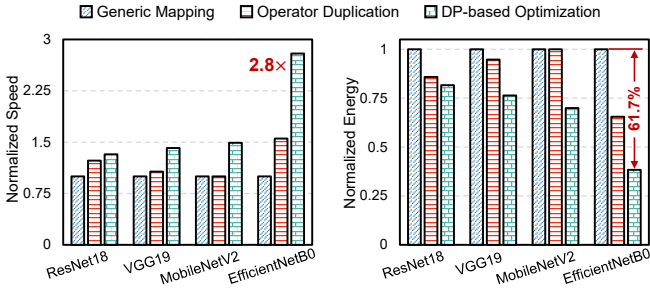
Fig. 5. Normalized speed and energy comparison of different compilation optimization strategies across DNN models.
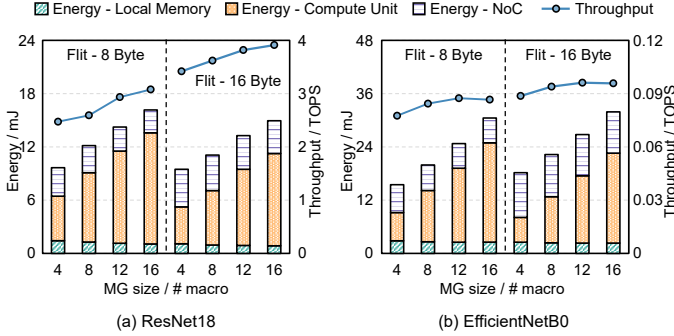

Fig. 6. Energy consumption breakdown and throughput across architectures with different MG sizes and NoC link bandwidth.

Fig. 7. Software/Hardware design space categorized by MG size. Light/dark shades indicate 8/16-byte NoC flit sizes.

operator duplication technique from CIM-MLC [19], which first partitions CG to fit the limited capacity then attempts to utilize vacant resources through weight duplication. All evaluations use the default architecture configuration described in Tab. I to isolate the impact of compilation strategies.

As illustrated in Fig. 5, our DP-based partitioning and optimization method demonstrates significant performance improvements, achieving up to 2.8× speedup and 61.7% energy reduction compared to the baseline approaches. The benefits are particularly pronounced for compact models like MobileNetV2 and EfficientNetB0, where the conventional partition method proves less effective due to their smaller weight footprints, which leaves fewer unoccupied cores within such partitioned execution stages for duplication opportunities. This highlights the effectiveness of our DP-based approach in finding optimal partitioning and mapping schemes that maximize performance while respecting SRAM capacity constraints.

C. Architectural Configuration Exploration

The efficiency of digital CIM architectures hinges on the careful balance between computational and data movement capabilities. We explore this trade-off through two critical design parameters: MG size scaling and NoC link bandwidth (flit size per cycle) configuration. Fig. 6 presents the energy breakdown and throughput analysis across architectural configurations for both compute-intensive and compact models compiled with the generic mapping strategy.

For ResNet18, increasing MG size consistently improves throughput at the cost of moderately higher energy consumption, with compute unit energy remaining its dominant com-
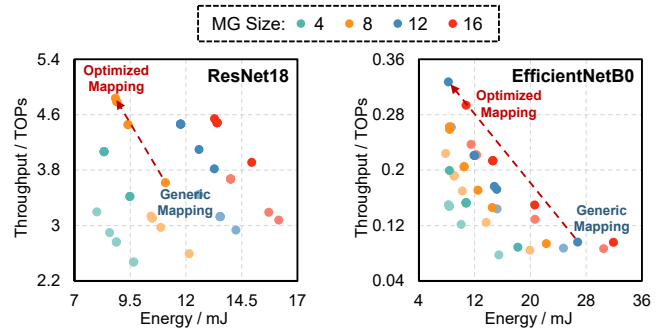
ponent. Doubling the communication bandwidth also boosts inter-layer pipeline throughput by up to 39.6%. In contrast, EfficientNetB0 shows different scaling characteristics. Its lower resource requirements mean that increasing the number of macros per group yields only modest throughput gains, while higher NoC bandwidth introduces substantial data transfer overhead without commensurate performance benefits, consuming up to 55.4% of total energy consumption when MG size is 4. These distinct scaling behaviors across models highlight the importance of early-stage architectural exploration, demonstrating the value of CIMFlow as a systematic design and evaluation framework.

To provide insights into the interaction between software and hardware design choices, we further compare different compilation strategies across these hardware configurations. As shown in Fig. 7, while hardware configurations significantly impact the achievable performance envelope, the performance differences between hardware configurations can be significantly reduced or even reversed through careful compilation optimizations. These observations highlight why an integrated hardware-software co-design approach is essential for digital CIM architectures, as isolated exploration of either space would overlook crucial optimization opportunities.

V. CONCLUSION AND FUTURE WORK

This paper presents CIMFlow, an integrated framework that enables systematic design and evaluation of digital CIM architectures. Through a flexible multi-level ISA design and advanced compilation strategies, CIMFlow effectively addresses key challenges in digital CIM implementation, particularly the SRAM capacity limitations. Our experimental results demonstrate the capabilities of our proposed compilation optimizations, achieving up to 2.8× speedup and 61.7% energy reduction. The highly customizable and extensible nature of CIMFlow enables systematic design space exploration, providing crucial insights for the software and hardware design of digital CIM. In the future, we will keep on expanding the framework to support emerging DNN operators and developing automated design space exploration techniques. We believe CIMFlow represents a significant step toward making digital CIM a practical solution for next-generation AI accelerators.

REFERENCES

[1] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits (JSSC)*, vol. 52, no. 1, pp. 127–138, 2016.

[2] J. Yang, W. Fu, X. Cheng, X. Ye, P. Dai, and W. Zhao, "S^2 Engine: A novel systolic architecture for sparse convolutional neural networks," *IEEE Transactions on Computers (TC)*, vol. 71, no. 6, pp. 1440–1452, 2021.

[3] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "AI and memory wall," *IEEE Micro*, vol. 44, no. 3, pp. 33–39, 2024.

[4] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 27–39, 2016.

[5] Y. Zhao, J. Yang, B. Li, X. Cheng, X. Ye, X. Wang, X. Jia, Z. Wang, Y. Zhang, and W. Zhao, "NAND-SPIN-based processing-in-MRAM architecture for convolutional neural network acceleration," *Science China Information Sciences (SCIS)*, vol. 66, no. 4, p. 142401, 2023.

[6] C. Duan, J. Yang, X. He, Y. Qi, Y. Wang, Y. Wang, Z. He, B. Yan, X. Wang, X. Jia *et al.*, "DDC-PIM: Efficient algorithm/architecture co-design for doubling data capacity of SRAM-based processing-in-memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 43, no. 3, pp. 906–918, 2023.

[7] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[8] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy *et al.*, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems (ASPLOS)*, 2019, pp. 715–731.

[9] J.-W. Su, Y.-C. Chou, R. Liu, T.-W. Liu, P.-J. Lu, P.-C. Wu, Y.-L. Chung, L.-Y. Hung, J.-S. Ren, T. Pan *et al.*, "16.3 a 28nm 384kb 6T-SRAM computation-in-memory macro with 8b precision for AI edge chips," in *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64. IEEE, 2021, pp. 250–252.

[10] Y.-D. Chih, P.-H. Lee, H. Fujiwara, Y.-C. Shih, C.-F. Lee, R. Naous, Y.-L. Chen, C.-P. Lo, C.-H. Lu, H. Mori *et al.*, "16.4 an 89TOPS/W and 16.3 TOPS/mm 2 all-digital SRAM-based full-precision compute-in memory macro in 22nm for machine-learning edge applications," in *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 252–254.

[11] B. Yan, J.-L. Hsu, P.-C. Yu, C.-C. Lee, Y. Zhang, W. Yue, G. Mei, Y. Yang, Y. Yang, H. Li *et al.*, "A 1.041-Mb/MM 2 27.38-TOPS/W signed-int8 dynamic-logic-based ADC-less SRAM compute-in-memory macro in 28nm with reconfigurable bitwise operation for AI and embedded applications," in *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 188–190.

[12] C. Duan, J. Yang, Y. Wang, Y. Wang, Y. Qi, X. He, B. Yan, X. Wang, X. Jia, and W. Zhao, "Towards efficient sram-pim architecture design by exploiting unstructured bit-level sparsity," in *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*, 2024, pp. 1–6.

[13] Q. Zheng, X. Li, Y. Guan, Z. Wang, Y. Cai, Y. Chen, G. Sun, and R. Huang, "PIMulator-NN: An event-driven, cross-level simulation framework for processing-in-memory-based neural network accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 12, pp. 5464–5475, 2022.

[14] H. Liu, J. Xu, X. Liao, H. Jin, Y. Zhang, and F. Mao, "A simulation framework for memristor-based heterogeneous computing architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 12, pp. 5476–5488, 2022.

[15] J. Lee, A. Lu, W. Li, and S. Yu, "Neurosim v1. 4: Extending technology support for digital compute-in-memory toward 1nm node," *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS-I)*, vol. 71, no. 4, pp. 1733–1744, 2024.

[16] Z. Zhu, H. Sun, T. Xie, Y. Zhu, G. Dai, L. Xia, D. Niu, X. Chen, X. S. Hu, Y. Cao *et al.*, "MNSIM 2.0: A behavior-level modeling tool for processing-in-memory architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 11, pp. 4112–4125, 2023.

[17] A. Siemieniuk, L. Chelini, A. A. Khan, J. Castrillon, A. Drebes, H. Corporaal, T. Grosser, and M. 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 (TCAD)*, vol. 41, no. 6, pp. 1674–1686, 2021.

[18] X. Sun, X. Wang, W. Li, L. Wang, Y. Han, and X. Chen, "PIMCOMP: A universal compilation framework for crossbar-based PIM DNN accelerators," in *Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[19] S. Qu, S. Zhao, B. Li, Y. He, X. Cai, L. Zhang, and Y. Wang, "CIM-MLC: A multi-level compilation stack for computing-in-memory accelerators," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024, pp. 185–200.

[20] C.-J. Jhang, C.-X. Xue, J.-M. Hung, F.-C. Chang, and M.-F. Chang, "Challenges and trends of SRAM-based computing-in-memory for AI edge devices," *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS-I)*, vol. 68, no. 5, pp. 1773–1786, 2021.

[21] J. Chen, F. Tu, K. Shao, F. Tian, X. Huo, C.-Y. Tsui, and K.-T. Cheng, "AutoDCIM: An automated digital CIM compiler," in *Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[22] F. Tu, Y. Wang, Z. Wu, L. Liang, Y. Ding, B. Kim, L. Liu, S. Wei, Y. Xie, and S. Yin, "ReDCIM: Reconfigurable digital computing-in-memory processor with unified FP/INT pipeline for cloud AI acceleration," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 58, no. 1, pp. 243–255, 2022.

[23] G. Desoli, N. Chawla, T. Boesch, M. Avodhyawasi, H. Rawat, H. Chawla, V. Abhijith, P. Zambotti, A. Sharma, C. Cappetta *et al.*, "16.7 A 40-310TOPS/W SRAM-based all-digital up to 4b in-memory computing multi-tiled NN accelerator in FD-SOI 18nm for deep-learning edge applications," in *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2023, pp. 260–262.

[24] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[25] *IEEE Standard for Standard SystemC® Language Reference Manual*, IEEE Std., 2023, IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011).

[26] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Noxim: An open, extensible and cycle-accurate network on chip simulator," in *Proceedings of the 26th IEEE international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2015, pp. 162–163.