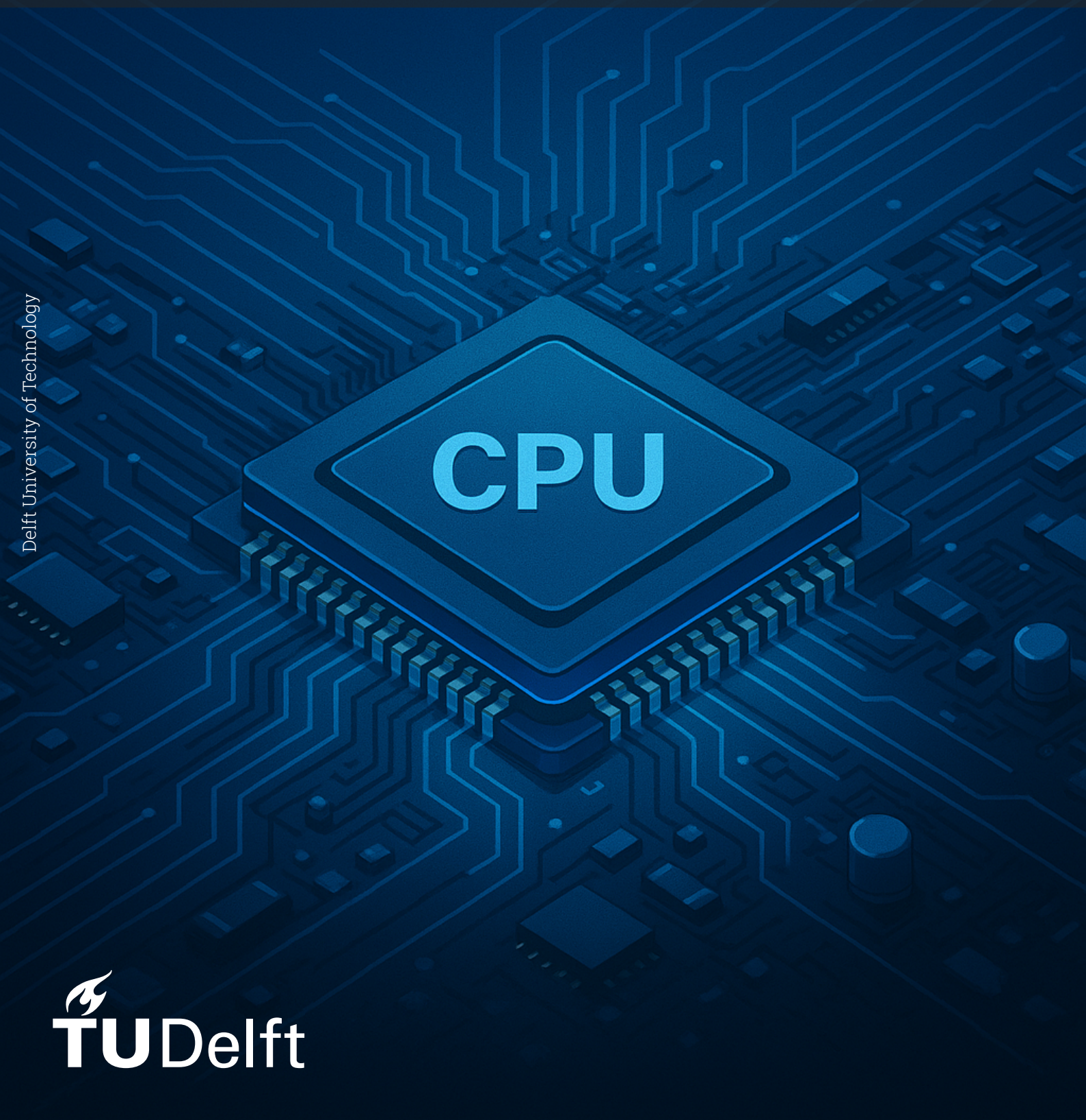
Simulating traffic for CIM accelerators

Fan Zhou



Delft University of Technology

TU Delft

Simulating traffic for CIM accelerators

by

Fan Zhou

to obtain the degree of Master of Science
at the Delft University of Technology

Student number: 601639
Project Duration: October, 2024 - August, 2025
Thesis committee:

Prof.dr.ir.G.N.Gaydadjiev,	TU Delft, supervisor
Prof.dr.ir.H.X.Lin,	TU Delft
T.Spyrou PhD,	TU Delft, daily supervisor

TU Delft

Preface

It has been a challenging journey to complete this thesis, and I could not have accomplished it entirely on my own. I would like to take this opportunity to sincerely thank all the people who have supported me, both technically and emotionally.

First and foremost, I would like to express my deepest gratitude to my supervisors, Prof. Georgi Gaydadjiev, Theofilos Spyrou, and Ignacio Garcia Ezquerro, for their invaluable guidance and support. Georgi, in particular, provided constant encouragement, insightful feedback, and expert advice that were instrumental throughout this research. I also thank Konstantinos Stavrakakis and Mahmood Naderan-Tahan from Delft University of Technology (NL) for their helpful discussions, guidance, and constructive comments on my work and thesis writing.

Also, I would like to thank my friends, both inside and outside the university.

Finally, I am deeply grateful to my parents for their unwavering emotional support and inspiration throughout this journey.

To all of you, thank you sincerely!

Fan Zhou
Delft, August 2025

Abstract

The rapid advancement of artificial intelligence (AI) and deep learning has intensified computational demands, exposing inefficiencies in traditional von Neumann architectures due to the ”memory wall” problem. Computation-in-Memory (CIM) emerges as a promising paradigm, performing computations directly within memory arrays to minimize data movement and enhance energy efficiency. However, current CIM design methodologies face significant challenges in configuration generation, performance characterization, and application-specific optimization. This thesis addresses these gaps by proposing a simulation framework that enables architectural exploration of multi-tile CIM-based NN accelerators with a focus on network traffic modeling, resource utilization, and communication efficiency.

The framework abstracts NN operations into hardware-executable patterns, encompassing both fully connected and convolutional layers, and incorporates key CIM components such as crossbars, accumulators, and activation units. Specialized mapping strategies and tiling techniques are introduced to capture realistic execution, while a bandwidth-constrained interconnect model quantifies communication bottlenecks. The framework supports both major convolution-to-MVM conversion schemes, Im2Col and K2M, allowing systematic evaluation of trade-offs between latency, area, and energy efficiency. Additionally, the framework integrates a topology visualization module that automatically generates Graphviz-based diagrams of component interconnections and data flows, enabling intuitive inspection of communication patterns and design bottlenecks.

Experimental evaluation on MNIST demonstrates the framework’s capability to reveal fundamental design insights. Results show that while K2M maximizes crossbar utilization (0.8–1.0) and reduces latency by up to 100×, Im2Col achieves up to 1000× reductions in crossbar count and data transfer energy, making it favorable for area- and energy-constrained systems. Bandwidth analysis further highlights the interaction between communication capacity and optimal crossbar sizing, establishing design guidelines that balance delay, efficiency, and scalability.

Overall, this thesis contributes a flexible and extensible simulation environment for multi-tile CIM accelerators that bridges the gap between neural network workloads and CIM hardware design, providing a foundation for future architectural exploration and hybrid optimization strategies in NN accelerators.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

Abbreviation	Definition
AI	artificial intelligence
BLAS	Basic Linear Algebra Subprograms
CIM	computation in memory
CNN	convolution neural network
Conv	convolution
CPU	central processing unit
DL	deep learning
DRAM	dynamic random-access memory
FC	fully connected
FCNN	fully connected neural network
GPU	Graphics processing unit
MAC	multiply-accumulate
MVM	matrix-vector multiplication
NN	neural network
NN accelerators	neural network accelerators
NoC	network on chip
PCM	phase-change memory
ReRAM	resistive random-access memory
TOPS	trillion operations per second

1

Introduction

This chapter introduces the topic addressed in this manuscript. Section 1.1 analyzes the computational demands of modern AI systems and the limitations of von Neumann architectures that necessitate compuation-in-memory(CIM) solutions, particularly highlighting on-chip communication bottlenecks. Section 1.2 identifies three critical gaps in current multi-tile CIM design tools: the lack of systematic mapping methodologies between neural networks and crossbar arrays, insufficient performance quantification capabilities, and absence of visualization tools for accelerator topologies. The thesis objectives in Section 1.3 proposes a framework with automated configuration generation, precise performance characterization, and advanced visualization tools to address these gaps. Section 1.4 delineates the study's boundaries by focusing on multi-tile CIM systems for inference of fully connected and convolution neural networks while abstracting certain physical characteristics. Key contributions in Section 1.5 include a modeling framework, evaluation framework, and topology visualization tools that advance CIM accelerator methodology. Finally, Section 1.6 provides the thesis organization.

1.1. Motivation

The rapid advancement of artificial intelligence (AI), particularly in deep learning(DL), has created unprecedented computational demands. Modern neural networks, such as vision transformers [8] with billions of parameters, and complex tasks like real-time video analysis require massive computational throughput [4]. Traditional computing architectures, rooted in the Von Neumann model [33], have seen significant advancements over time [17, 12]. However, they still struggle to meet these demands due to the ”memory wall” [29] problem, a fundamental bottleneck where energy and latency of the computer systems are dominated by data movement between physically separated processing units (e.g., CPUs, GPUs) and memory hierarchies (DRAM, caches) [41]. This inefficiency is exacerbated in data-intensive workloads. For example, large language models (LLMs), which have surged in popularity because of their powerful generative capabilities that far surpass earlier state-of-the-art methods, pose significant challenges in terms of computation, particularly the intensive compute requirements and high energy costs associated with inference [37].

To overcome this limitation, Computation-in-Memory (CIM) [13] has emerged as a promising paradigm, where computation is performed directly within the memory arrays, minimizing data transfers. CIM architectures, such as SRAM-based Processing-in-Memory (PIM) [15] and memristor crossbars [3], have demonstrated significant improvements in energy efficiency and throughput for matrix-vector operations, a fundamental kernel in neural networks(NNs). However, as CIM systems scale to accommodate larger neural networks, multi-tile communication between and within neural network layers emerges as a critical performance bottleneck.

This underscores the urgent need for specialized design tools that can:

- Accurately model data traffic patterns in multi-tile CIM systems;
- Explore network-on-chip (NoC) architectures optimized for CIM-specific communication patterns;
- Quantify tradeoffs between computational density, communication latency, and energy efficiency.

Our work addresses this gap by developing a cycle-accurate simulation framework that enables systematic exploration of interconnect architectures for next-generation CIM accelerators.

1.2. Problem Statement

The design of multi-tile CIM neural network accelerators currently lacks comprehensive tools to bridge the gap between algorithmic requirements and hardware implementation constraints. Existing methodologies fail to provide solutions for three critical challenges in accelerator design:

First, there is no standardized approach to map neural network parameters (weights and activations) to multiple physical crossbar arrays while accounting for several architectural constraints. This mapping problem becomes increasingly complex when considering: (a) different neural network architectures (fully connected vs. convolution layers), each requiring distinct dataflow patterns; (b) variable bit precision requirements (from 1-bit binary to 8-bit fixed-point representations) that affect how weights are distributed across crossbars; (c) physical crossbar size limitations (currently 128×128 to 256×256 devices) that necessitate sophisticated tiling strategies; and (d) available bandwidth constraints that dictate how quickly activations and partial sums can be moved between computational units.

Second, existing frameworks lack the capability to accurately quantify system-level communication costs during complete neural network inferences. This includes simulating: (a) crossbar resource utilization efficiency, which determines how effectively the hardware resources are employed during computation; (b) precise accounting of bit transfers among components, which can help estimate total data movement energy; and (c) end-to-end computation latency, particularly the delays introduced by bandwidth limitations and component synchronization. Without these metrics, designers cannot perform meaningful trade-off analyses between different architectural choices.

Third, there is a notable absence of visualization tools that can provide intuitive representations of architectural topologies, particularly for designs combining heterogeneous computational units (e.g., crossbars, accumulators, and activation modules) with multi-bit-width data paths. This opacity made it difficult to validate basic connectivity, trace dataflow dependencies, or identify shared-resource contention during early design stages.

The absence of the above capabilities in current design tools forces researchers and engineers to make suboptimal architectural decisions, potentially leaving significant performance and energy efficiency gains unrealized. This problem becomes particularly acute as neural network models grow in complexity and CIM architectures scale to larger arrays of computational units. A simulation framework capable of modeling these limitations would significantly advance CIM accelerator design, enabling designers to optimize next-generation AI hardware more efficiently.

1.3. Research Objectives

This thesis aims to develop a comprehensive analytical framework that addresses the critical gaps in multi-tile CIM accelerator design through three primary technical objectives.

First, the framework will establish an automated methodology for generating multi-tile CIM accelerator configurations. This involves developing algorithms that intelligently determine the minimal set of required crossbar components and generate the component connection topologies by analyzing the neural network's computational demands, matrix-to-crossbar mapping strategies, and hardware constraints such as array dimensions and precision requirements.

Second, the framework will incorporate precise performance characterization capabilities to quantitatively evaluate key CIM configuration parameters, including crossbar size, on-chip network bandwidth, and bit precision specifications. Through simulation of complete neural network inference operations, it will track and record the following key metrics: (1) the number of activated CIM crossbars and their utilization rates, revealing computational resource efficiency; (2) the total volume of transferred bits across all interconnects, which directly reflects data movement energy consumption; and (3) the estimated processing delay that accounts for both data dependencies and bandwidth constraints. These measurements will provide designers with concrete insights into computational efficiency, energy costs, and timing characteristics of their CIM accelerator implementations.

Third, the framework will generate clear, structured visualizations of the accelerator's physical interconnect topology, explicitly mapping hierarchical connections, such as data distribution from the host

through multi-bit-width crossbars, and highlighting critical transitions between heterogeneous components. By automating this foundational visualization, the tool will empower designers to efficiently verify architectural correctness, debug interconnect mismatches, and gain immediate insight into the full system architecture. This high-level overview will enable preliminary assessment of bandwidth bottlenecks and dataflow efficiency, serving as a crucial first step before detailed dynamic simulation.

Together, these objectives will produce a naive design framework that bridges the gap between neural network algorithms and CIM hardware implementation, enabling more efficient and automated development of next-generation AI accelerators. The framework will specialize in co-optimizing neural network architectures with physical hardware parameters, enabling designers to navigate trade-offs between resource utilization, computational latency, and energy efficiency.

1.4. Scope

This thesis focuses on developing a multi-tile CIM accelerator simulation framework with carefully defined boundaries to enable targeted investigation of communication bottlenecks in CIM systems. The scope encompasses four key dimensions of the framework's capabilities and limitations.

The simulation framework specifically targets two fundamental types of neural network layers: fully connected (FC) layers and convolution layers. These architectures were selected because they represent the most common building blocks of many deep learning systems while demonstrating distinct computational patterns - FC layers exhibit regular matrix-vector operations ideal for crossbar implementations, while CNNs require specialized handling of sliding-window convolution operations. The framework currently excludes more complex layer types such as recurrent networks or attention mechanisms to maintain focus on core computational patterns.

Regarding operational characteristics, the simulation framework exclusively models inference operations using pre-trained and quantized neural network models. This design choice reflects the predominant use case for CIM accelerators in edge computing and embedded applications where energy-efficient inference is paramount. The framework assumes networks have already undergone quantization-aware training, with weights properly discretized to match the target hardware's precision capabilities. Training operations are explicitly excluded due to their fundamentally different computational patterns and memory access requirements.

The hardware modeling approach employs abstracted but configurable representations of crossbar arrays as fundamental computational units. Users can specify key parameters including crossbar dimensions (typically 128×128 for current digital implementations) and bit precision (fixed-point with configurable width), while the simulation framework handles the mapping of neural network operations to these constrained resources. The model intentionally simplifies certain physical characteristics by assuming idealized memory access behavior without endurance or retention issues, allowing researchers to isolate and study communication effects separate from device-level non-idealities.

For traffic analysis, the simulation framework operates at two complementary levels of abstraction. At the component level, the framework monitors key operational metrics, including active crossbar count and utilization rates, delivering granular insight into hardware resource efficiency. System-level analysis captures aggregate behavior through total transferred bits and total cycle count for end-to-end inference, which serve as proxies for energy consumption and throughput. The framework deliberately excludes certain physical effects, such as analog noise, thermal variations, and detailed wire parasitics, to maintain a focus on digital communication patterns and enable reproducible baseline comparisons.

This carefully scoped approach enables tractable verification of core simulation framework functionality using the MNIST dataset and relatively simple neural network architectures. The choice of MNIST reflects its well-understood characteristics and modest computational requirements, which facilitate rapid iteration and debugging before scaling to more complex models. While limited in immediate scope, the framework is designed to provide foundational insights into communication bottlenecks that will inform future extensions to more sophisticated networks and hardware models. The current implementation prioritizes clear, reproducible results that establish performance baselines for digital CIM architectures while maintaining extensibility for future research directions.

1.5. Contributions

This thesis makes several significant contributions to the field of multi-tile CIM accelerator design through the development of a network traffic simulation framework.

- **Parameterized framework**: the thesis proposes a framework that systematically bridges the gap between neural network algorithms and its multi-tile CIM hardware implementations. The framework accepts comprehensive input specifications including neural network architecture, bit precision requirements, crossbar array dimensions, and available bandwidth constraints. Based on these parameters, it automatically generates hardware configurations by determining the minimal number of required crossbar components, implementing efficient matrix-to-crossbar mapping strategies, and generating detailed topological representations of inter-unit connection patterns.

  Quantitative evaluation framework: the thesis develops a suite of complementary performance metrics that extend beyond conventional measures to capture resource efficiency, energy proxies, and latency in multi-tile CIM accelerators. By integrating these metrics into the proposed simulation environment, the framework enables fine-grained analysis of crossbar utilization, comprehensive accounting of hardware resources, and detailed data-movement tracking. These capabilities allow designers to explore and compare architectural trade-offs with unprecedented precision, offering deeper insight into performance bottlenecks and guiding more informed design optimizations.
- **Topology visualization**: this thesis further contributes a design exploration tool that significantly accelerates early-stage accelerator development through automated topology visualization. The framework generates clear, structured representations of the physical interconnect architecture, explicitly mapping hierarchical connections. By providing this intuitive visualization of component interconnections and bandwidth characteristics, the tool enables designers to rapidly verify architectural correctness, identify potential interconnect mismatches, and gain immediate insight into system-level dataflow efficiency. These capabilities offer a crucial first-step evaluation of the accelerator's static structure, allowing preliminary bottleneck assessment and informed optimization decisions before committing to detailed simulation.

By integrating these capabilities, this work makes important methodological advances in several key areas. It establishes a systematic approach for mapping logical neural network operations to physical CIM resources, accounting for both algorithmic requirements and hardware constraints. The framework introduces quantitative evaluation techniques that provide objective measures of architectural effectiveness, moving beyond simplistic performance estimates. The visualization tools transform abstract accelerator designs into intuitive topological representations, enabling designers to immediately grasp critical dataflow relationships and perform preliminary optimization.

These contributions are particularly valuable during the critical system-level design phase when neural network characteristics are known but hardware implementation decisions must still be made. The framework provides essential guidance for navigating the complex design space where computational requirements, hardware constraints, and performance targets must be carefully balanced. By addressing this crucial phase in accelerator development, the thesis helps bridge the persistent gap between neural network algorithms and efficient hardware realization in multi-tile CIM systems. The resulting framework not only advances academic understanding of multi-tile CIM design principles but also provides practical tools that can accelerate the development of next-generation AI accelerators.

1.6. Thesis Outline

The rest of this report is organized into size chapters. Chapter 2 is providing a background and related works of CIM. The next two chapters are explaining the methodology and implementation of the proposed simulation framework. Then, the manuscript uses one chapter to analyze the experimental results of the simulation framework. Finally, the last chapter concludes this thesis. The following list provides a more detailed description of the topics discussed in each chapter:

Chapter 2 introduces the background of CIM and points two difficulties of applying CIM to neural network processing. Finally, it presents two methods that can potentially assist CIM to accelerate the convolution operation.

Chapter 3 explains the methodology used to model the multi-tile CIM accelerator that processes the neural network in a structured and organized manner.

Chapter 4 describes the implementation of the simulation framework, which is based on the model from the last chapter.

Chapter 5 proposes two neural network architecture and analyzes the results obtained from simulating these two neural networks using the proposed simulation framework.

Chapter 6 summarizes the thesis and discusses potential future work.

2

Background & Related Works

This chapter presents the fundamental concepts and state-of-the-art approaches for Computation-in-Memory (CIM) neural network accelerators. Section 2.1 introduces the CIM paradigm, detailing its two primary architectural implementations (analog and digital) and their respective advantages for neural network processing. Section 2.2 examines the critical challenges in mapping neural networks to CIM hardware, including matrix-to-crossbar mapping strategies and dataflow optimization techniques. The core focus of the chapter, Section 2.2.3, analyzes two principal methods for converting convolution operations to matrix-vector multiplications: the Im2Col approach (Section 2.2.3) and the K2M method (Section 2.2.3). These techniques are compared in Section 2.2.3 through quantitative analysis of their memory requirements, computational efficiency, and hardware implementation characteristics.

2.1. Computation-in-memory (CIM)

2.1.1. Definition

Computation-in-Memory (CIM) is a non-von Neumann computing paradigm that eliminates the physical separation between memory and processing units. Unlike traditional architectures where data shuttles between discrete CPU/GPU cores and memory hierarchies (e.g., DRAM, caches), CIM leverages the physical properties of memory devices to perform computations directly within the memory array itself. This approach exploits two fundamental mechanisms: (1) parallel analog computation in resistive memory technologies (e.g., memristors, ReRAM), where Ohm's Law [31] (I = V×G) and Kirchhoff's Law [32] enable multiply-accumulate (MAC) operations through current summation; and (2) digital bitwise operations in modified SRAM/DRAM cells, where additional logic gates are embedded within memory subarrays. These mechanisms give rise to two distinct architectures: analog and digital CIM architecture.

2.1.2. Current CIM Archtitectures

1. Analog CIM Architectures

   Analog CIM exploits the native physics of emerging non-volatile memory devices to perform computations in the analog domain [2]. Resistive crossbars, made up of memristors [11], ReRAM [42], or phase-change memory (PCM) [39] devices, encode weights as conductance values (G) and inputs as voltages (V), generating output currents (I) that naturally compute MVM operations via Ohm's law. Mythic AI's analog matrix processors [30] and IBM's HERMES project [19] showcase how such crossbars achieve O(1) complexity for MAC operations, delivering 100–1,000 TOPS/W efficiency. However, analog systems face challenges including device variability (5–10% conductance drift), thermal noise, and ADC/DAC overheads. Recent advances like hybrid precision architectures (e.g., ISAAC [6]) combine analog crossbars with digital post-processing to mitigate these issues. Ferroelectric FET (FeFET)-based CIM is another promising direction [26], offering non-destructive reads and high endurance ($> 10^{12}$ cycles), though maturity lags behind resistive memories.

2. Digital CIM Architectures

   Digital CIM designs integrate programmable logic elements directly into conventional memory technologies, maintaining compatibility with standard CMOS processes. SRAM-based Processing-in-

Memory (PIM) architectures, such as Samsung's HBM-PIM [38], augment memory banks with simple ALUs to enable bitwise operations (AND, OR, XOR) and integer arithmetic within DRAM subarrays. These designs excel at deterministic operations and are backward-compatible with existing software stacks, but suffer from area overhead due to added logic. UPMEM's DRAM-PIM [23] takes an alternative approach by embedding RISC-V [40] cores directly into memory banks, offering flexibility for diverse workloads but incurring higher power consumption. Recent research, such as the ComputeDRAM framework, demonstrates how bitline computing in DRAM can perform bulk bitwise operations without logic modifications, though precision is limited to 1–4 bits. Digital CIM is particularly suited for edge devices requiring moderate precision (INT8/INT16) and deterministic execution.

2.1.3. Advantage of CIM

By colocating computation and storage, CIM reduces energy consumption greatly compared to von Neumann systems, primarily by minimizing data movement. This paradigm is transformative for data-intensive workloads like NNs, where weight matrices can be permanently stored in memory arrays and activated by input vectors without off-chip transfers. In general, the key benefits of CIM include:

- Energy Efficiency: Eliminates redundant data movements;
- Parallelism: Native support for bulk operations (e.g., MVM);
- Scalability: Memory-centric design avoids bandwidth bottlenecks.

2.2. CIM for Neural Networks

2.2.1. Matrix-to-Crossbar Mapping

The efficient mapping of neural network parameters to physical crossbar arrays is a critical challenge in CIM accelerator design. For large weight matrices that exceed the dimensions of a single crossbar (typically 128×128 for current resistive arrays), tiling strategies partition the matrix into smaller sub-blocks that can be distributed across multiple crossbars. This approach introduces interconnect overhead but preserves the inherent parallelism of CIM architectures. When dealing with multi-bit precision weights (e.g., 4-8 bits), bit-slicing techniques decompose the weight matrix into separate crossbars, each handling a specific bit position. This enables higher-precision computations while maintaining analog efficiency, though it requires careful synchronization during partial sum accumulation.

2.2.2. Dataflow Strategies

The choice of dataflow strategy significantly impacts both energy efficiency and computational throughput in CIM neural network accelerators. In weight-stationary architectures, weight matrices are permanently programmed into crossbar conductances, while input activations are streamed through the array. This approach minimizes weight movement energy (which dominates in CNNs) but requires careful scheduling of activation transfers. Alternatively, output-stationary designs keep partial sums local to each crossbar during computation, reducing the need for intermediate sum transfers at the cost of increased crossbar reprogramming overhead. Modern CIM accelerators often employ hybrid dataflows; for example, using weight-stationary mapping for convolution layers (where weights are reused across sliding windows) while adopting output-stationary approaches for fully connected layers (where partial sums exhibit less reuse). The optimal strategy depends on both the neural network architecture and the underlying CIM technology's reprogramming latency and energy characteristics.

2.2.3. Convolution-to-MVM Conversion Methods

Convolution neural networks (CNNs) [20] fundamentally rely on 2D convolution operations between input feature maps and weight kernels. However, Computation-in-Memory (CIM) architectures natively accelerate matrix-vector multiplication (MVM) operations. This architectural mismatch necessitates reformulating convolution operations as MVM problems to fully leverage CIM's parallel computing capabilities. The conversion process addresses three key challenges: (1) preserving the sliding-window nature of convolutions, (2) maintaining spatial relationships between pixels, and (3) efficiently mapping the computation to fixed-size crossbar arrays. Without such conversion, CIM architectures would require inefficient sequential processing of convolution windows, negating their energy and speed advantages. Currently, there are two main coversion methods, Im2Col and K2M.

Im2Col [16]
The Im2Col method transforms the convolution operation into a single large matrix multiplication by unrolling all possible convolution windows from the input feature map. For an input tensor $XR^{H\times W\times C}$ (height × width × channels) and K kernels of size $F \times F \times C$, the method:

1. Extracts all $F \times F \times C$ patches from X with stride S;
2. Flattens each patch into a column vector of length F^2C;
3. Constructs matrix $X'R^{(F^2C)\times(OH\times OW)}$ where OH, OW are output dimensions;
4. Reshapes kernels into matrix $WR^{K\times(F^2C)}$.

The convolution becomes: $Y = W \times X'$

Where $YR^{K(OHOW)}$ is the output feature map. This approach effectively converts the convolution into a single MVM operation compatible with CIM crossbars. However, it introduces memory overhead due to input duplication - the transformed matrix X' is larger than the original input by a factor of F^2/S^2.

Example To clarify this method, a specific example will be employed. Consider an input tensor $X \in \mathbb{R}^{3\times3\times1}$ (single channel) and a 2×2 kernel with stride 1:

1. **Input Transformation**:
   - Original input (1 channel):
   $$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$
   ;
   - Extracted 2×2 patches (stride 1):
   $$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}, \begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}, \begin{bmatrix} 4 & 5 \\ 7 & 8 \end{bmatrix}, \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}$$
   ;
   - Flattened into columns ($F^2C = 4$ elements per column):
   $$X' = \begin{bmatrix} 1 & 2 & 4 & 5 \\ 2 & 3 & 5 & 6 \\ 4 & 5 & 7 & 8 \\ 5 & 6 & 8 & 9 \end{bmatrix}^\top \in \mathbb{R}^{4\times4}$$
   .
2. **Kernel Transformation**:
   - Original kernels (K_1 and K_2):
   $$K_1 = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad K_2 = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$
   ;
   - Flattened and stacked as rows:
   $$W = \begin{bmatrix} a & b & c & d \\ e & f & g & h \end{bmatrix} \in \mathbb{R}^{2\times4}$$
   .
3. **Matrix Multiplication**: The convolution becomes:
   $$Y = W \times X' = \begin{bmatrix} a\cdot1 + b\cdot2 + c\cdot4 + d\cdot5 & \cdots \\ e\cdot1 + f\cdot2 + g\cdot4 + h\cdot5 & \cdots \end{bmatrix} \in \mathbb{R}^{2\times4}$$
   Each row of Y corresponds to one kernel's output flattened.

This approach enables efficient MVM operations but duplicates input elements (e.g., value 5 appears 4 times in X').

K2M [1]

The K2M method reformulates convolution by constructing a *Toeplitz matrix* [14] from the kernel weights, which is then multiplied with the flattened input. Unlike Im2Col, this approach preserves the original input structure while expanding the kernel into a sparse matrix.

Example To clarify this method, a specific example will be employed. Consider an input $X \in \mathbb{R}^{3\times3\times1}$ and a single 2×2 kernel with stride 1:

1. **Original Input and Kernel**:

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}, \quad K = \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix}$$

2. **Flattened Input**:

$$\text{vec}(X) = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{21} & x_{22} & x_{23} & x_{31} & x_{32} & x_{33} \end{bmatrix}^\top \in \mathbb{R}^{9\times1}$$

3. **Kernel Matrix Construction**: The kernel is transformed into a sparse matrix $W \in \mathbb{R}^{4\times9}$ (for output size 2×2) with:

$$W = \begin{bmatrix} k_{11} & k_{12} & 0 & k_{21} & k_{22} & 0 & 0 & 0 & 0 \\ 0 & k_{11} & k_{12} & 0 & k_{21} & k_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & k_{11} & k_{12} & 0 & k_{21} & k_{22} & 0 \\ 0 & 0 & 0 & 0 & k_{11} & k_{12} & 0 & k_{21} & k_{22} \end{bmatrix}$$

   Each row corresponds to one position in the output feature map, with the kernel weights positioned according to the input pixels they multiply.

4. **Matrix Multiplication**: The convolution is computed as:

$$Y = W \times \text{vec}(X) = \begin{bmatrix} k_{11}x_{11} + k_{12}x_{12} + k_{21}x_{21} + k_{22}x_{22} \\ k_{11}x_{12} + k_{12}x_{13} + k_{21}x_{22} + k_{22}x_{23} \\ k_{11}x_{21} + k_{12}x_{22} + k_{21}x_{31} + k_{22}x_{32} \\ k_{11}x_{22} + k_{12}x_{23} + k_{21}x_{32} + k_{22}x_{33} \end{bmatrix} \in \mathbb{R}^{4\times1}$$

The key features of K2M are as follows:

- K2M creates a very sparse matrix where each row contains exactly F^2 non-zero elements;
- The matrix size grows with both input size ($H \times W$) and output size ($OH \times OW$);
- Unlike Im2Col, the original input is used without duplication.

Comparison and Tradeoffs

Table 2.1 summarizes the key differences between the K2M and Im2Col methods for converting convolution operations into matrix–vector multiplications. The two approaches mainly differ in which tensor is transformed: K2M expands the kernel into a Toeplitz matrix, whereas Im2Col expands the input into overlapping patches.

In terms of matrix shape, K2M produces a matrix of size $(OH \times OW) \times (HWC)$, while Im2Col yields a $(F^2C) \times (OH \times OW)$ matrix. These layouts directly impact memory overhead: K2M requires storing a large, highly sparse matrix, while Im2Col duplicates input elements, resulting in a dense but potentially larger input representation.

Another important distinction is how the original data is modified. In K2M, the input tensor remains preserved, but in Im2Col the input is unrolled and duplicated. This makes K2M more favorable for small kernels with large inputs, where sparsity can be exploited, while Im2Col is advantageous for large kernels with small inputs, where duplication overhead is relatively lower.

From a hardware perspective, K2M demands sparse optimizations to avoid wasted computation on zero entries, while Im2Col directly benefits from dense Basic Linear Algebra Subprograms(BLAS) [7] routines and existing accelerator support.

Overall, K2M trades off storage for sparsity, whereas Im2Col trades off duplication for compatibility with dense computation. The choice between them depends on kernel size, input dimensions, and available hardware optimizations.

Table 2.1: Key Differences Between K2M and Im2Col

Aspect	**K2M**	**Im2Col**
Transformed Element	Kernel (Toeplitz matrix)	Input (patches)
Matrix Shape	$(OH \times OW) \times (HWC)$	$(F^2C) \times (OH \times OW)$
Memory Overhead	Large sparse matrix	Duplicated input
Sparsity	Highly sparse	Dense
Input Modification	Original preserved	Unrolled/duplicated
Best For	Small kernels, large inputs	Large kernels, small inputs
Hardware Friendliness	Needs sparse optimizations	Works with dense BLAS

3

Modeling Methodology

This chapter establishes a systematic modeling framework for computation-in-memory (CIM) accelerators from top to bottom, focusing on hardware-algorithm co-design principles. Section 3.1 details the foundational matrix-vector multiplication (MVM) operations in resistive crossbars, covering matrix mapping strategies (Section 3.1.3), input/output handling, and tiling techniques for large matrices. Section 3.2 presents a novel method for multi-bit MVM using single-bit crossbars, including data representation (Section 3.2.2), crossbar organization (Section 3.2.3), computation phases (Section 3.2.4), and traffic analysis (Section 3.2.5). Section 3.3 abstracts neural network layers into hardware-executable patterns, covering fully connected (Section 3.3.1), convolution (Section 3.3.2), and pooling/flattening layers (Section 3.3.3). Section 3.4 provides component-level models for crossbars (Section 3.4.1), accumulators (Section 3.4.2), activation units (Section 3.4.3), and pooling/flattening modules (Section 3.4.4). Finally, Section 3.5 develops a bandwidth-constrained communication model, incorporating practical interconnect limitations for enhanced model precision. Section 4.7 consludes the chapter.

3.1. Matrix-Vector Multiplication

In-memory computing using resistive crossbars enables highly efficient matrix-vector multiplication (MVM) by leveraging Ohm's Law for current summation and Kirchhoff's Law for column-wise accumulation. To perform MVM operations, the matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ must be properly mapped onto the crossbar array, where each conductance value G_{ij} of a memristive device at row i and column j represents the matrix element W_{ij}, while the vector is treated as the input of the crossbar.

3.1.1. Matrix Mapping

A typical $m \times n$ crossbar consists of m wordlines (rows) and n bitlines (columns), with memristive devices at each intersection. The matrix $\mathbf{W}$ is mapped such that:

$$G_{ij} \propto W_{ij}, \tag{3.1}$$

where G_{ij} is the conductance of the device at location (i, j).

3.1.2. Input Vector Application and Output Readout

The input vector $\mathbf{x} \in \mathbb{R}^n$ is applied as analog voltages V_j to the bitlines, scaled to avoid device nonlinearity. The resulting currents along each wordline are summed vertically due to Kirchhoff's Law, producing an output current vector $\mathbf{I} \in \mathbb{R}^m$:

$$I_i = \sum_{j=1}^{n} G_{ij} V_j. \tag{3.2}$$

These currents are then converted to digital values, yielding the MVM result $\mathbf{y} = \mathbf{W}\mathbf{x}$.

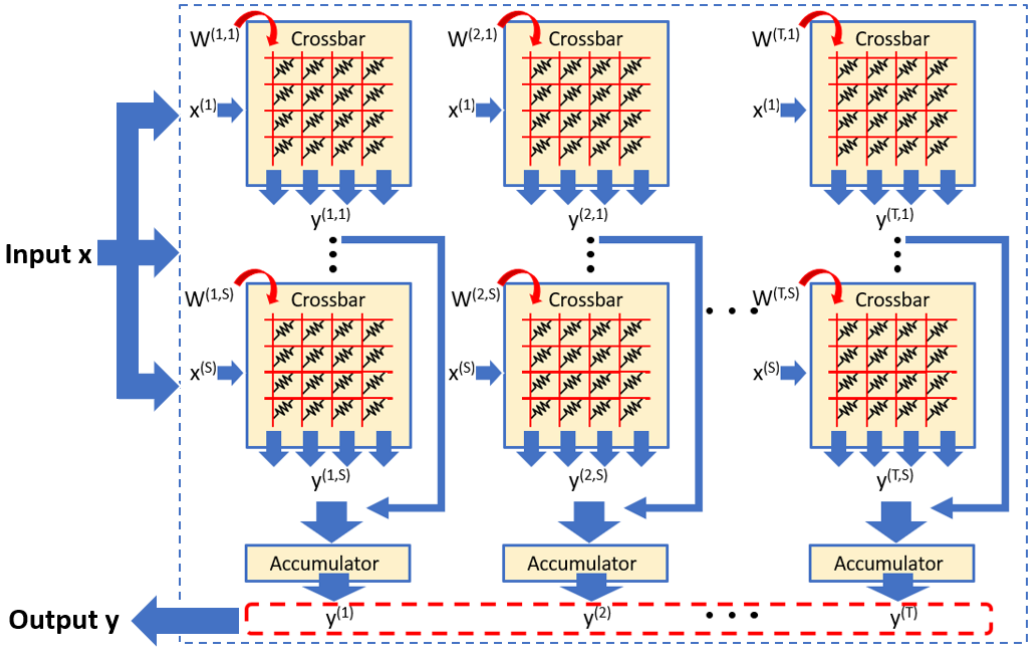


Figure 3.1: Matrix-vector multiplication (MVM) using crossbar tiling.

3.1.3. Matrix Tiling for Large-Scale Matrices

The dimensions of the matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ will most probably exceed the physical size of a crossbar array ($p \times q$, where $p < m$ and/or $q < n$). In this case, the matrix must be partitioned into smaller submatrices (tiles) that fit within the crossbar, as shown in Fig. 3.1. The input vector $\mathbf{x}$ is split into segments $(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(S)})$, and the weight matrix $\mathbf{W}$ is partitioned into tiles $(\mathbf{W}^{(1,1)}, \ldots, \mathbf{W}^{(T,S)})$. Each tile is mapped to a crossbar, which computes partial results $(\mathbf{y}^{(1,1)}, \ldots, \mathbf{y}^{(T,S)})$. These partial results are accumulated in order to produce the final outputs $(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(T)})$. Zero-padding may be applied to ensure divisibility.

Matrix Partitioning Strategy

Let $\mathbf{W}$ be divided into $T \times S$ tiles, where each tile $\mathbf{W}^{(t,s)} \in \mathbb{R}^{p \times q}$ is mapped to a separate crossbar operation. The partitioning is computed as:

$$T = \left\lceil \frac{m}{p} \right\rceil, \quad S = \left\lceil \frac{n}{q} \right\rceil, \tag{3.3}$$

with zero-padding added to the matrix edges if m or n is not divisible by p or q. For simplicity, assume $m = T \cdot p$ and $n = S \cdot q$ hereafter. The input vector $\mathbf{x} \in \mathbb{R}^n$ is similarly split into S segments $\mathbf{x}^{(s)} \in \mathbb{R}^q$.

Crossbar Computation and Output Accumulation

Each tile $\mathbf{W}^{(t,s)}$ computes a partial MVM with its corresponding input segment $\mathbf{x}^{(s)}$, producing an intermediate output $\mathbf{y}^{(t,s)} \in \mathbb{R}^p$:

$$\mathbf{y}^{(t,s)} = \mathbf{W}^{(t,s)} \mathbf{x}^{(s)}. \tag{3.4}$$

To reconstruct the full output $\mathbf{y} \in \mathbb{R}^m$, the partial results are summed along the column partitions ($s = 1, \ldots, S$) for each row partition ($t = 1, \ldots, T$):

$$\mathbf{y}^{(t)} = \sum_{s=1}^{S} \mathbf{y}^{(t,s)}, \quad \mathbf{y} = \begin{bmatrix} \mathbf{y}^{(1)} \\ \vdots \\ \mathbf{y}^{(T)} \end{bmatrix}. \tag{3.5}$$

3.2. Multi-Bit MVM Using Single-Bit Crossbars

3.2.1. Problem Statement

The manuscript focuses on simulating the data traffic of multi-tile CIM accelerator with single-bit crossbars. Thus, there exists a problem, how to use single-bit crossbars to implement multi-bit MVM. The

multi-bit MVM computation requires:

- Matrix $\mathbf{W} \in \mathbb{Z}^{m \times n}$ with k-bit signed precision;
- Input vector $\mathbf{x} \in \mathbb{Z}^n$ with k-bit signed precision;
- Single-bit crossbar arrays performing bit-wise operations.

3.2.2. Data Representation

1. **Matrix Decomposition**: Each w_{ij} is expanded to its k-bit two's complement representation:

$$w_{ij} = \sum_{b=0}^{k_w-2} 2^b w_{ij}^b \tag{3.6}$$

   where $w_{ij}^b \in \{0, 1\}$.

2. **Input Vector Expansion**: Each input x_j is decomposed similarly:

$$x_j = \sum_{a=0}^{k_x-2} 2^a x_j^a \tag{3.7}$$

   and represented as a $k \times n$ binary matrix $\mathbf{X}_{\text{bit}}$.

3.2.3. Crossbar Organization

The crossbar-based computing fabric integrates analog crossbar arrays for massively parallel bit-wise dot-product operations and digital peripheral circuits for weighted accumulation and precision management. This hybrid organization leverages the strengths of both domains:

- Analog crossbars efficiently perform large-scale Boolean multiplications in parallel through Ohm's and Kirchhoff's laws;
- Digital circuitry provides precise accumulation, bit-position alignment, and synchronization across computation phases.

This architecture enables multi-bit matrix-vector multiplication by decomposing both weights and inputs into their binary components, which are then orchestrated through a combination of static mapping for weights and sequential application for inputs.

Matrix Mapping

The $m \times n$ matrix $\mathbf{W}$ with k-bit precision is transformed into an expanded binary representation:

$$\mathbf{W}_{\text{bin}} = \begin{pmatrix} w_{11}^{k-1} & \cdots & w_{11}^0 & \cdots & w_{1n}^{k-1} & \cdots & w_{1n}^0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ w_{m1}^{k-1} & \cdots & w_{m1}^0 & \cdots & w_{mn}^{k-1} & \cdots & w_{mn}^0 \end{pmatrix} \tag{3.8}$$

where $\mathbf{W}_{\text{bin}}$ has dimensions $m \times (n \cdot k)$. This expanded matrix is directly mapped to the crossbar array such that:

- Each original element w_{ij} occupies k consecutive columns;
- The most significant bit (MSB) w_{ij}^{k-1} is placed in the leftmost column of its group;
- The least significant bit (LSB) w_{ij}^0 occupies the rightmost column.

Input Vector Mapping

The n-element input vector $\mathbf{x}$ with k-bit precision undergoes a similar transformation:

$$\mathbf{X}_{\text{bin}} = \begin{pmatrix} x_1^{k-1} & \cdots & x_n^{k-1} \\ \vdots & \ddots & \vdots \\ x_1^0 & \cdots & x_n^0 \end{pmatrix} \tag{3.9}$$

The input process requires k temporal phases:

1. In phase p ($0 \leq p < k$), the p-th bit-slice $\mathbf{X}_{\text{bin}}[p, :]$ is applied to all columns;
2. Each phase completes in one unit time;
3. The complete input sequence requires k unit times.

The crossbar organization ensures that:

- Matrix bits remain static during computation;
- Input bits are applied sequentially;
- Physical column adjacency matches logical bit significance.

3.2.4. Computation Process

The matrix-vector multiplication operation is executed through a carefully orchestrated sequence of analog and digital computations. The complete process consists of three distinct phases that transform binary-weighted inputs and weights into precise arithmetic results.

Bit-wise Multiplication Phase

The fundamental computation occurs at the crossbar arrays where single-bit weights interact with single-bit inputs. For each input bit position a ($0 \leq a < k$) and weight bit position b ($0 \leq b < k$), the crossbar do the computation:

$$p_{ij}^{ab} = w_{ij}^{b} \wedge x_{j}^{a} \tag{3.10}$$

This Boolean multiplication is physically implemented through:

- **Matrix representation**: Memristor conductances G_{ij}^{b} programmed to:

$$G_{ij}^{b} = \begin{cases} G_{\text{on}} & \text{if } w_{ij}^{b} = 1 \\ G_{\text{off}} & \text{if } w_{ij}^{b} = 0 \end{cases} \tag{3.11}$$

- **Input application**: Voltage pulses $V_{j}^{a}(t)$ applied to columns:

$$V_{j}^{a}(t) = \begin{cases} V_{\text{read}} & \text{if } x_{j}^{a} = 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

- **Current sensing**: Output currents I_{i}^{ab} at row i represent the logical AND operation:

$$I_{i}^{ab} \propto \sum_{j=1}^{n} G_{ij}^{b} V_{j}^{a}(t) \tag{3.13}$$

Bit-Position Scaling

The raw bit products require significance weighting according to their positional values. Each product p_{ij}^{ab} is scaled by:

$$s_{ij}^{ab} = 2^{a+b} \cdot p_{ij}^{ab} \tag{3.14}$$

It can be implemented by scaling voltage-domain.

$$V_{j}^{a}(t) = 2^{a} \cdot V_{\text{ref}} \cdot x_{j}^{a} \tag{3.15}$$

Result Accumulation

The final output is generated through a hierarchical accumulation process:

1. **Crossbar-level summation**:

$$y_{i}^{ab} = \sum_{j=1}^{n} s_{ij}^{ab} \tag{3.16}$$

   performed inherently through Kirchhoff's current law at each row node.

2. **Digital weight accumulation**:
$$y_i^a = \sum_{b=0}^{k-1} 2^b \cdot y_i^{ab} \tag{3.17}$$
implemented through shift-add operations in peripheral digital circuitry.
3. **Input bit accumulation**:
$$y_i = \sum_{a=0}^{k-1} 2^a \cdot y_i^a \tag{3.18}$$
completed in the output accumulator over k clock cycles.

The computation process maintains strict synchronization between the analog crossbar operations and digital accumulations, with timing controlled by a central controller that manages:

- Input bit-slice application sequence;
- ADC sampling windows;
- Digital accumulation cycles.

3.2.5. Data Traffic and Transfer Delay Analysis

Data Volume Characterization

For an $m \times n$ weight matrix with k-bit precision and n-element k-bit input vector:

- **Crossbar Input**:
$$\mathcal{T}_{\text{in}} = k \cdot n \text{ bits} \tag{3.19}$$
(Serial transfer of n elements $\times$ k bit-slices);
- **Crossbar Output**:
$$\mathcal{T}_{\text{out}} = k \cdot (m \cdot k) \text{ bits} \tag{3.20}$$
(For m output lines $\times$ k input bits $\times$ k weight bits);
- **Accumulator Input**: Matches crossbar output:
$$\mathcal{T}_{\text{acc-in}} = \mathcal{T}_{\text{out}} = k^2 m \text{ bits} \tag{3.21}$$
;
- **Accumulator Output**:
$$\mathcal{T}_{\text{acc-out}} = k \cdot m \text{ bits} \tag{3.22}$$
(Final k-bit outputs for m elements).

Transfer Timing Analysis

Define τ_{unit} as the system clock period, equal to the maximum of:

$$\tau_{\text{unit}} = \max(\tau_{\text{bit}}, \tau_{\text{cycle}}, \tau_{\text{transfer}}) \tag{3.23}$$

where:

- τ_{bit}: Input bit application time;
- τ_{cycle}: Crossbar computation cycle;
- τ_{transfer}: Data Transfer Time Between Components.

The data transfer timeline proceeds as:

1. **Input Phase** (k cycles):
$$\tau_{\text{input}} = k \cdot \tau_{\text{unit}} \tag{3.24}$$
2. **Crossbar to Accumulator Transfer** (k cycles):
$$\tau_{\text{cross-out}} = k \cdot \tau_{\text{unit}} \tag{3.25}$$
(Pipelined with computation)

Total transfer delay (excluding computation):

$$\tau_{\text{transfer-total}} = 2k \cdot \tau_{\text{unit}} \tag{3.26}$$

Bandwidth Requirements

The sustained bandwidth between components must satisfy:

- **Crossbar Output Bus**:

$$BW_{\text{crossbar}} \geq m \cdot k \cdot \frac{1}{\tau_{\text{unit}}} \tag{3.27}$$

;

- **Accumulator Input Bus**:

$$BW_{\text{acc}} \geq m \cdot k \cdot \frac{1}{\tau_{\text{unit}}} \tag{3.28}$$

.

3.2.6. Motivational Example

This section presents a concrete example of signed matrix-vector multiplication using 2-bit precision, accompanied by detailed data traffic and timing analysis.

System Specifications

The example system consists of:

- A 2×2 matrix using signed 2's complement representation;
- 2-bit precision for both weights and inputs (numerical range: -2 to 1);
- Crossbar array dimensions: 2×4 (2 rows $\times$ 2 weights $\times$ 2 bits).

Binary Representations

The matrix $\mathbf{W}$ and input vector $\mathbf{x}$ are represented in 2's complement binary format:

$$\mathbf{W} = \begin{pmatrix} 1 & 1 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 01_2 & 01_2 \\ 00_2 & 10_2 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} 3 \\ 0 \end{pmatrix} = \begin{pmatrix} 11_2 \\ 00_2 \end{pmatrix} \tag{3.29}$$

where the most significant bit (MSB) represents the sign.

Data Traffic Analysis

The computation involves two primary data transfer phases, quantified in Table 3.1:

Table 3.1: Data Transfer Characteristics

Transfer Phase	Bits Transferred	Cycles	Description
Input Loading	4	2	Serial transfer of 2 input elements $\times$ 2 bits
Crossbar Output	8	2	Parallel transfer of 2 rows $\times$ 4 bits/row
Total	12	4	

Result Validation

The computation proceeds as follows:

Digital Reference:

$$y_1 = 1 \times 3 + 1 \times 0 = 3$$
$$y_2 = 0 \times 3 + 2 \times 0 = 0$$

Crossbar Outputs:

MSB Cycle: $(0, 1, 0, 1)$

LSB Cycle: $(0, 1, 0, 1)$

Accumulation:

$$y_1 = 2^2 \times 0 + 2^1 \times (0 + 1) + 2^0 \times 1 = 3$$
$$y_2 = 2^2 \times 0 + 2^1 \times (0 + 0) + 2^0 \times 0 = 0$$

3.3. Neural Network Layer Modeling

The simulator abstracts neural network layers into hardware-executable patterns, focusing on four fundamental layer types that collectively cover most CNN and Fully Connected Neural Network architectures, including fully connected layer, convolution layer, pooling layer and flatten layer. Each layer type is modeled with distinct computational characteristics and hardware resource requirements.

3.3.1. Fully Connected Layer Architecture

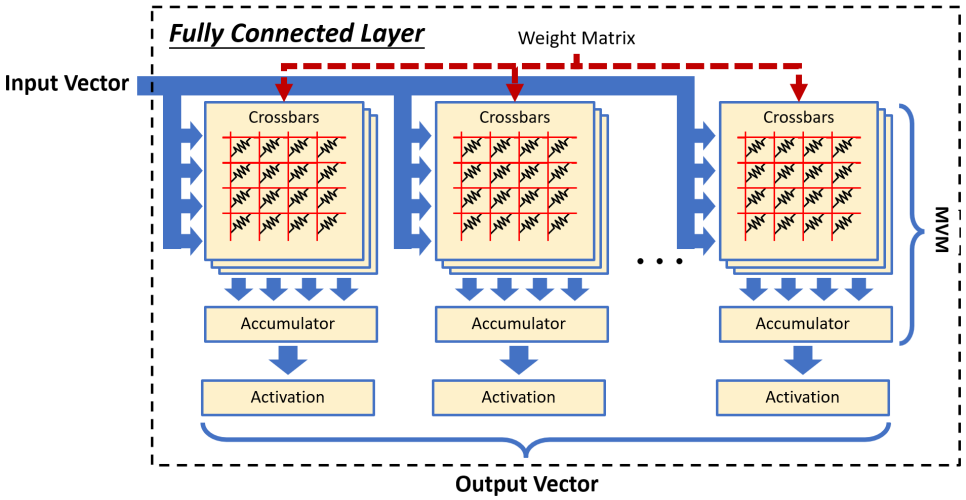


Figure 3.2: Architecture of the fully connected layer in the CIM accelerator.

The fully connected (FC) layer is a fundamental component of neural networks, where each neuron connects to all activations from the previous layer. In the proposed **Computation-in-Memory (CIM) accelerator**, the FC layer is implemented using resistive crossbar arrays to perform efficient matrix-vector multiplication (MVM) operations.

Mathematical Formulation

For an input vector $\mathbf{x} \in \mathbb{R}^n$ and weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$, the output $\mathbf{y} \in \mathbb{R}^m$ is computed as:

$$\mathbf{y} = \phi\left(\mathbf{W} \cdot \mathbf{x} + \mathbf{b}\right) \tag{3.30}$$

where $\phi(\cdot)$ is the activation function, and $\mathbf{b} \in \mathbb{R}^m$ is the bias vector. Based on the formulation, the computation of FC layer can be abstracted into three stages:

1. Analog MVM in crossbar arrays to get partial products;
2. Digital accumulation (summing partial products to get final MVM results);
3. Nonlinear activation.

Structure of the FC Layer

As illustrated in Fig. 3.2, the FC layer's implementation decomposes into the following key components, each of which will be modeled in detail in Section 3.4:

1. **Crossbar Arrays**:
   - *Function:* Perform MVM in analog domain via Ohm's/Kirchhoff's laws.
   - *Weight Mapping:* Weights are stored in conductance values of resistive memory devices (e.g., memristors). The precise mapping methodology for the weight matrix onto the crossbar arrays is detailed in Section 3.1.3.
   - *Input Handling:* Input vectors are applied as voltage signals, and Ohm's Law/Kirchhoff's Law compute the dot product in analog domain. For large input dimensions, the input vector is partitioned and distributed across multiple crossbars, as described in Section 3.1.3.
2. **Accumulators**:

- *Function:* Partial sums from crossbar columns are aggregated digitally.
- *Non-Ideality Mitigation:* Resolve non-idealities (e.g., sneak paths, device variations) via calibration circuits.

3. **Activation Unit**:
   - *Function:* Applies nonlinear functions (e.g., ReLU, Sigmoid) to the accumulated outputs.

This three-stage abstraction—crossbars, accumulators, and activation—enables a modular analysis of the FC layer's performance and trade-offs, which will be explored in Section 3.4.

3.3.2. Convolution Layer Architecture

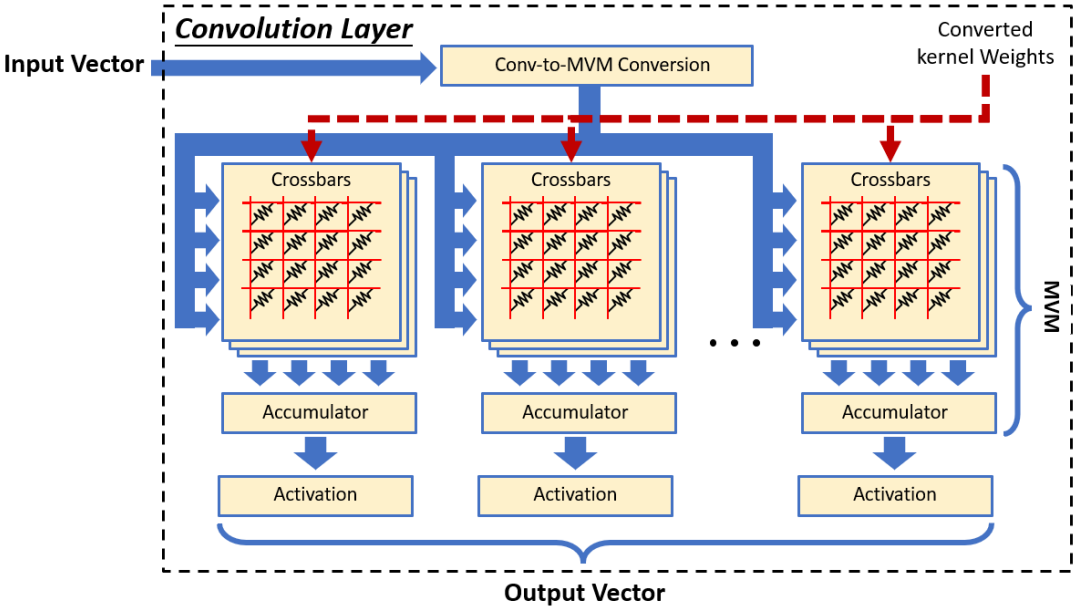


Figure 3.3: Architecture of the convolution layer in the CIM accelerator.

The convolution (Conv) layer performs spatial feature extraction through localized filter operations. In the proposed **Computation-in-Memory (CIM) accelerator**, the Conv layer computation is abstracted into matrix-vector multiplication (MVM) operations compatible with the crossbar infrastructure used in FC layers (Sec. 3.3.1).

Mathematical Formulation

For an input feature map $\mathbf{X} \in \mathbb{R}^{H\times W\times C}$ and kernel $\mathbf{W} \in \mathbb{R}^{K\times K\times C\times F}$, the output $\mathbf{Y} \in \mathbb{R}^{H'\times W'\times F}$ can be computed through either of two formulations due to different conversion methods(Section 2.2.3):

1. `Im2Col` method:

$$\mathbf{Y} = \phi(\mathbf{W} \cdot \texttt{Im2Col}(\mathbf{X}) + \mathbf{b}) \tag{3.31}$$

2. `K2M` (Kernel-to-Matrix) method:

$$\mathbf{Y} = \phi(\texttt{reshape}(\mathbf{W}) \cdot \mathbf{X}_{\text{flat}} + \mathbf{b}) \tag{3.32}$$

where $\phi(\cdot)$ is the activation function, and $\mathbf{b} \in \mathbb{R}^F$ is the bias vector. Based on these formulations, the computation can be abstracted into:

- `Im2Col` path: Four stages (input reshaping + three FC-equivalent stages);
- `K2M` path: Three stages (identical to FC layer).

Structure of the Conv Layer

As illustrated in Fig. 3.3, the Conv layer implementation varies by conversion method:

1. **Conversion Unit** (for `Im2Col` only):

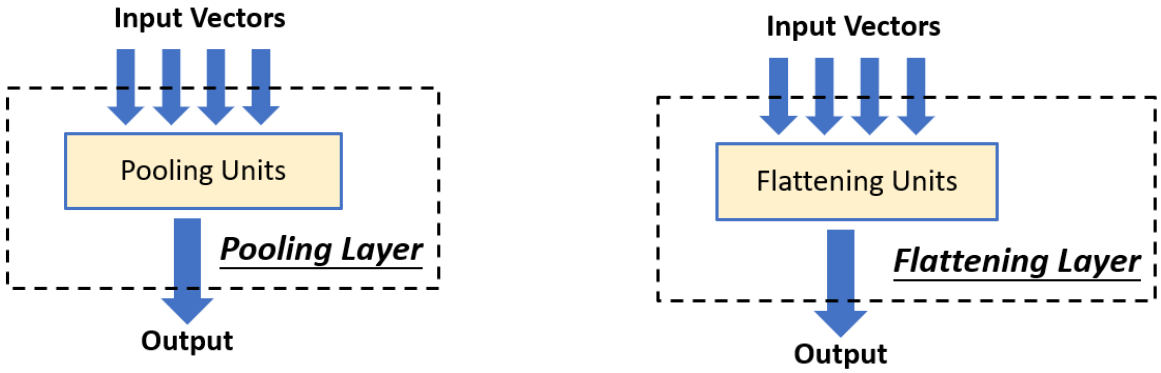
- *Function:* Dynamically reshapes input feature maps using the `Im2Col` algorithm.
- *Bypass Condition:* Not required for `K2M` method where kernels are pre-reshaped.

2. **Crossbar Arrays**:
   - *Function*: Performs matrix-vector multiplication (MVM) operations using the same fundamental principles as the FC layer implementation (Section 3.3.1).
   - *Weight Mapping*:
     - For the `K2M` method, kernel weights are pre-reshaped into Toeplitz matrices during the programming phase and stored in conductance values;
     - For the `Im2Col` approach, each kernel is flattened into column vectors and mapped to crossbar columns.
   - *Input Handling*:
     - Processes either `Im2Col`-transformed input patches or flattened feature maps, depending on the conversion method employed;
     - Incorporates stride and padding parameters during the input reshaping process to maintain spatial correspondence.
3. **Accumulators**:
   - *Function:* Identical to FC layer implementation (Sec. 3.3.1).
4. **Activation Unit**:
   - *Function:* Identical to FC layer implementation.

This abstraction demonstrates that the Conv layer either:

- Adds one preprocessing component (`Im2Col` unit) while reusing all FC layer components;
- Requires no additional components when using `K2M` conversion.

3.3.3. Pooling and Flattening Layer Architectures



(a) Pooling layer abstraction.

(b) Flattening layer abstraction.

Figure 3.4: Hardware abstraction of (a) pooling and (b) flattening layers in the CIM accelerator.

The pooling and flattening layers represent simpler operations in neural networks that require distinct hardware handling compared to FC and convolution layers. In the proposed **Computation-in-Memory (CIM) accelerator**, these layers are abstracted as single functional units to maintain architectural consistency while optimizing for their specific computational patterns.

Mathematical Formulation

- **Pooling Layer**: For an input feature map $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$ and pooling window size P, the output $\mathbf{Y} \in \mathbb{R}^{H' \times W' \times C}$ is computed as:

$$\mathbf{Y}_{i,j,c} = \text{pool}(\mathbf{X}_{iP:(i+1)P, jP:(j+1)P, c}) \tag{3.33}$$

  where $\text{pool}(\cdot)$ represents max, average, or other pooling operations;

- **Flattening Layer**: For an input $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$, the output $\mathbf{y} \in \mathbb{R}^{HWC}$ is simply:

$$\mathbf{y} = \text{vec}(\mathbf{X}) \tag{3.34}$$

  where $\text{vec}(\cdot)$ denotes vectorization.

Structure of the Layers
As shown in Fig. 3.4, both layers are abstracted by dedicated functional units:

- **Pooling Unit**:
  - Implements certain pooling operations(max or average in most cases) spatially across input feature maps;
  - Handles stride and padding parameters through configurable windowing logic;
  - Typically operates in digital domain after analog-to-digital conversion of inputs.
- **Flattening Unit**:
  - Reorganizes multi-dimensional input tensors into one-dimensional vectors;
  - Requires only routing logic without computational elements;
  - Maintains dimensional ordering consistency for subsequent FC layers.

The simplified abstraction of these layers as single components reflects their fundamental difference from MVM-based layers, while preserving the overall accelerator's modular design philosophy.

3.4. Component-Level Modeling

Since the manuscript focuses exclusively on bit-level traffic analysis within the CIM accelerator, all hardware components, including the CIM crossbar arrays, accumulators, activators, and specialized units for pooling and flattening, are modeled as traffic generators [27]. In this framework: 1. Each component receives bits from preceding units, processes them, and transmits bits to subsequent stages. 2. All transferred bits and delays are recorded and aggregated to evaluate the overall communication traffic across the accelerator. This approach allows for a systematic analysis of data movement, enabling optimizations for bandwidth, latency, and energy efficiency in CIM-based neural network processing.

3.4.1. CIM Crossbar Arrays

Computation-in-Memory (CIM) crossbar arrays utilize memristor-based crossbar architectures to perform analog matrix-vector multiplication (MVM), offering significant improvements in throughput and energy efficiency for neural network computations. In this work, we model the crossbar as a functional unit characterized by its input-output bitwidth constraints and computational latency, abstracting away the underlying physical implementation details.

Dataflow and Timing Model: Consider a weight matrix $\mathbf{W} \in \mathbb{R}^{M \times N}$ mapped to the crossbar array and an input matrix $\mathbf{X} \in \mathbb{R}^{P \times M}$. The computation proceeds through the following pipelined stages:

1. **Input Phase:** During each clock cycle t, one M-bit column vector $\mathbf{x}_t \in \mathbb{R}^M$ from the input matrix is streamed into the crossbar;
2. **Computation Phase:** The crossbar performs analog MVM operation $\mathbf{y}_t = \mathbf{W}\mathbf{x}_t$, generating an intermediate N-bit result that undergoes analog-to-digital conversion (ADC);
3. **Output Phase:** The quantized N-bit output vector $\hat{\mathbf{y}}_t$ is transmitted to subsequent processing units in cycle $t + 1$.

The total computation latency is determined by the input matrix dimensions:

$$\mathcal{T}_{\text{total}} = P \cdot \tau_{\text{cycle}},$$

where P represents the number of column vectors in the input matrix and τ_{cycle} denotes the duration of one clock cycle. The complete computation produces an output matrix $\mathbf{Y} \in \mathbb{R}^{P \times N}$.

This model assumes perfect pipeline utilization, with new inputs being processed and corresponding outputs being generated in each clock cycle until all P input vectors have been computed.

3.4.2. Accumulator Module

The accumulator processes partial results from CIM crossbar arrays through bit-precision and/or spatial accumulation, producing final computational outputs. The module handles two orthogonal accumulation scenarios that may occur simultaneously:

Accumulation Modes

1. **Bit-Precision Accumulation**: For p-bit inputs ($p > 1$), accumulates p bit-sliced components as described in Section 3.2;
2. **Spatial Accumulation**: Combines partial products from s crossbars processing matrix partitions:

$$\mathbf{y}_j = \sum_{k=1}^{s} \mathbf{a}_{k,j} \quad \forall j \in [1, n]$$

These modes operate concurrently when processing high-precision inputs on partitioned matrices, with the accumulation hierarchy being:

1. First aggregate spatial partitions within each bit slice;
2. Then combine bit-weighted results across slices.

Output Timing Characteristics

The accumulator's output phase exhibits deterministic latency:

- **Output Generation**: Requires 1 cycle after final input arrival to complete all accumulations;
- **Result Streaming**: Transmits p rows of n-bit results at 1 row/cycle:

$$\mathcal{T}_{\text{output}} = p \text{ cycles(pipelined with computation)} \tag{3.35}$$

The final output matrix $\mathbf{Y} \in \mathbb{R}^{p \times n}$ contains accumulated results, where p reflects either:

- The input bit precision in bit-sliced mode;
- The row dimension of spatial partitions;
- The product of both factors in combined operation.

3.4.3. Activation Function Module

The activation module applies element-wise nonlinear transformations to the output of the accumulator. The module maintains strict dimensional consistency between its input and output matrices while introducing the necessary nonlinearity for neural network operations.

Dataflow Characteristics

Given an input matrix $\mathbf{X} \in \mathbb{R}^{p \times n}$ from the accumulator:

- Processes one complete row per clock cycle;
- Applies activation function $f(\cdot)$ element-wise:

$$\mathbf{Y}_{i,j} = f(\mathbf{X}_{i,j}) \quad \forall i \in [1, p], j \in [1, n] \tag{3.36}$$

  where $f(\cdot)$ represents ReLU, sigmoid, or other activation functions.

Timing Behavior

The module exhibits perfectly linear throughput:

- **Latency**: Exactly matches the row dimension of the input;
- **Throughput**: Sustains 1 row/cycle processing rate.

The total processing time is given by:

$$\mathcal{T}_{\text{activation}} = p \text{ cycles} \tag{3.37}$$

where p is identical to the row dimension of both:

- The accumulator's output matrix;
- The activation module's output matrix $\mathbf{Y} \in \mathbb{R}^{p \times n}$.

3.4.4. Pooling Unit

The pooling unit performs spatial downsampling on multiple feature maps generated by the preceding convolution layer, reducing spatial dimensions while maintaining feature depth. The module operates on the principle of local receptive field aggregation with deterministic output sizing.

Dimensional Transformation

Given K input matrices $\mathbf{X}_k \in \mathbb{R}^{p \times n}$ from the convolution layer, where:

- $p = b \times r$ represents:
  - b: bit precision of computations;
  - r: number of output features.
- n: spatial dimension (width $\times$ height) per feature map.

For pooling kernel size $k \times k$ with stride s, the output dimension per feature map is:

$$n' = \left\lfloor \frac{\sqrt{n} - k}{s} + 1 \right\rfloor^2 \tag{3.38}$$

Dataflow Characteristics

- Processes K feature maps in parallel;
- Maintains row dimension p (bit precision $\times$ features);
- Reduces column dimension from n to n'.

Thus, the pooling unit dimensional transformation is shown in Table 3.2.

Table 3.2: Pooling Unit Dimensional Transformation

Parameter	Input	Output
Row Dimension	p	p
Column Dimension	n	n'
Number of Features	K	K

Timing Behavior

The module exhibits linear throughput characteristics:

- **Processing Rate**: 1 row per clock cycle
- **Total Latency**:

$$\mathcal{T}_{\text{pooling}} = p \text{ cycles} \tag{3.39}$$

3.4.5. Flattening Unit

The flattening unit transforms multi-dimensional feature maps from the convolution/pooling layers into a one-dimensional vector suitable for processing by fully-connected layers. The module performs a deterministic reshaping operation while preserving the numerical precision of the original data.

Dimensional Transformation

Given K input matrices $\mathbf{X}_k \in \mathbb{R}^{p \times n}$ from the preceding layer, where:

- $p = b \times r$ represents:
  - b: bit precision of computations;
  - r: number of output features.
- n: spatial dimension per feature map.

The flattening operation produces output vector $\mathbf{y} \in \mathbb{R}^{b \times m}$ where:

$$m = K \times r \times n \tag{3.40}$$

$$\mathbf{y} = \text{vec}\left(\begin{bmatrix} \mathbf{X}_1 & \mathbf{X}_2 & \cdots & \mathbf{X}_K \end{bmatrix}\right) \tag{3.41}$$

Dataflow Characteristics

- **Input Organization**:
  - Concatenates K feature maps column-wise;
  - Maintains bit-level organization across features.
- **Output Structure**:
  - Produces b output rows (bit precision);
  - Each row contains m elements (flattened features).

Timing Behavior

The unit operates with predictable latency:

- **Processing Rate**: 1 row per clock cycle
- **Total Latency**:

$$\mathcal{T}_{\text{flatten}} = b \text{ cycles} \tag{3.42}$$

Table 3.3: Flattening Unit Dimensional Transformation

Characteristic	Specification
Input Features	K
Input Rows per Feature	$p = b \times r$
Output Rows	b
Output Columns	$m = K \times r \times n$
Throughput	1 row/cycle

3.5. Bandwidth-Constrained Communication Model

While the models mentioned earlier often assume ideal data transfer conditions, practical implementations face significant bandwidth limitations that impact overall system performance. This section presents a detailed analytical model that captures these bandwidth constraints and their effects on neural network acceleration.

3.5.1. Model Motivation and Formulation

The bandwidth-constrained model addresses the limitation of ideal network-on-chip assumptions by incorporating three key realities:

1. Physical interconnect fabrics have finite bandwidth capacity;
2. Memory and processing components exhibit asymmetric I/O capabilities;
3. Data transfer often becomes the performance bottleneck in CIM systems.

The transmission delay between any two components follows the fundamental equation:

$$t_{\text{transmit}} = \left\lceil \frac{\text{size\_bits}}{bw_{\text{effective}}} \right\rceil \times \text{UNIT\_TIME} \tag{3.43}$$

where the parameters are defined as:

- size_bits: The data volume (in bits) for one row of the crossbar array;
- $bw_{\text{effective}}$: Achievable bandwidth (bits/cycle) considering all constraints;
- UNIT_TIME: The system's base clock cycle duration.

The effective bandwidth represents the tightest constraint in the communication path:

$$bw_{\text{effective}} = \min(bw_{\text{out}}, bw_{\text{in}}, bw_{\text{path}}) \text{ [28]} \tag{3.44}$$

comprising:

- bw_{out}: Source component's maximum output bandwidth;
- bw_{in}: Destination component's maximum input bandwidth;
- bw_{path}: Configured interconnect path bandwidth.

It should be noted that the current bandwidth model does not account for access conflicts. For instance, when multiple crossbars simultaneously transmit data to the same accumulator, the resulting contention introduces additional delays that are not captured in the present implementation. Addressing this limitation through enhanced conflict-aware bandwidth modeling is an important direction for future work.

3.5.2. Implementation Framework

The bandwidth management system operates through the precise algorithm shown in Algorithm 1, which implements Equations 3.43 and 3.44.

Algorithm 1 Bandwidth-Constrained Transmission

Input: Source address, destination address, data size (bits)
Output: Transmission delay (cycles)

$bw_{\text{out}} \leftarrow$ source.getOutPortBW()
$bw_{\text{in}} \leftarrow$ destination.getInPortBW()
$bw_{\text{path}} \leftarrow$ interconnect.queryPathBW(src, dest)
$bw_{\text{effective}} \leftarrow \min(bw_{\text{out}}, bw_{\text{in}}, bw_{\text{path}})$
$cycles \leftarrow \lceil \text{size\_bits} / bw_{\text{effective}} \rceil$
return $cycles \times$ UNIT_TIME {Convert to system time}

The implementation handles several practical considerations:

- **Discrete Time Quantization**: the ceiling function ensures conservative cycle counting;
- **Port Contention**: shared ports dynamically adjust available bandwidth;
- **Path Reservation**: critical paths can be allocated guaranteed bandwidth.

3.6. Conclusion

This chapter established a comprehensive modeling framework for multi-tile CIM accelerators. It first introduced the fundamental matrix–vector multiplication operations using resistive crossbars, then extended the discussion to multi-bit computation techniques with single-bit devices, incorporating tiling strategies for mapping large-scale matrices onto multi-tile CIM architectures. Building on the MVM-to-CIM theory, the thesis developed neural network layer abstractions and component-level models that enable efficient matrix-to-crossbar allocation and interconnect topology construction for neural network architectures. To further enhance realism, a bandwidth-constrained communication model was also proposed. Overall, the chapter lays a solid foundation for simulating and analyzing CIM-based neural network accelerators from device-level operations to system-level interconnects.

4

Implementation

This chapter details the implementation of proposed multi-tile CIM accelerator simulator from bottom to top, featuring a three-level hierarchy (components, layers, and networks) connected through a bandwidth-constrained interconnect. The chapter is organized as follows: Section 4.1 presents the system architecture and hierarchical organization. Section 4.2 details the component-level implementations. Section 4.3 describes the neural network layer compositions. Section 4.4 explains the complete network implementation, and Section 4.5 covers the critical interconnect subsystem. Section 4.6 explains the implementation of visualization function. Section 4.7 draws a brief conclusion for the chapter.

4.1. Implementation Architecture Overview

The data traffic simulator implements a hierarchical architecture centered around an interconnect fabric that facilitates communication between various neural network processing components, which is illustrated in Figure 4.1.

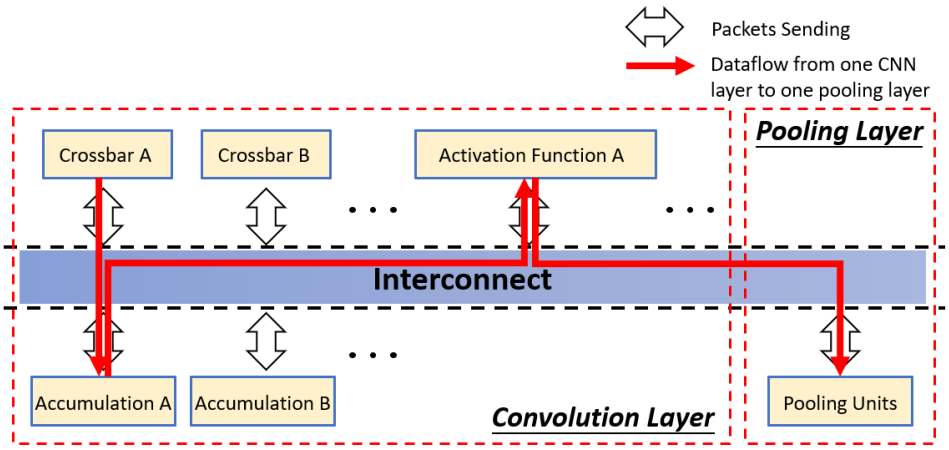


Figure 4.1: Interconnect architecture with example data flow (red).

4.1.1. System Hierarchy

The simulator architecture organizes neural network acceleration into three distinct hierarchical levels, as illustrated in Figure 4.2.

At the foundation lies the *Component Level*, comprising fundamental processing units including crossbars, accumulators, and activation functions. These primitive elements implement basic mathematical operations and maintain standardized interfaces for communication through the interconnect fabric.

The intermediate *Neural Network Layer Level* abstracts complete neural network layers by composing components into functional pipelines. Each layer encapsulates the sequence of operations required

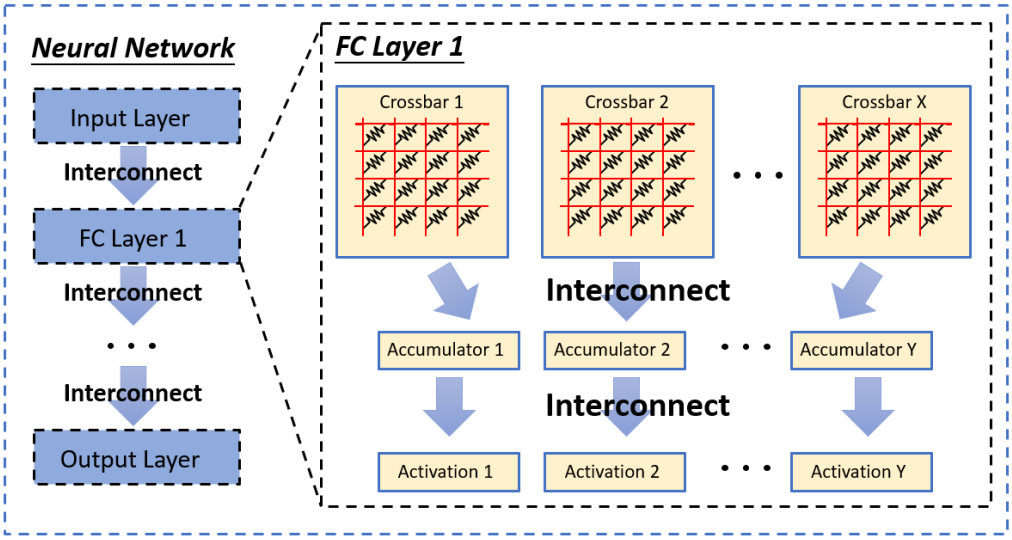


Figure 4.2: Component–layer–network hierarchy with interconnects.

for specific transformations, such as convolution or pooling operations, while maintaining clear input and output boundaries. The interconnect fabric serves as the critical infrastructure binding these layers together, ensuring proper data flow between successive layers.

At the highest abstraction level, the *Network Structure Level* integrates multiple neural network layers into complete processing pipelines. This tier manages the end-to-end data movement through the accelerator, from initial input processing to final output generation.

Throughout all three levels, the interconnect fabric performs dual roles: it facilitates intra-level communication between components within individual layers while simultaneously managing inter-level data transfers between successive layers' boundary components. This hierarchical organization enables modular analysis of computational efficiency at different abstraction levels while maintaining accurate modeling of system-wide data movement patterns.

4.1.2. Data Flow Mechanism

There is a representative data flow path highlighted in red in Figure 4.1, demonstrating the movement of data from crossbar components in a convolution layer to the subsequent pooling layer. It is a typical transition from convolution layer to pooling layer, which also reflects the data flow mechanism.

Initially, input data undergoes matrix multiplication operations within the crossbar arrays (Crossbar A/B), where the core linear algebra computations are performed. The raw computation results are then systematically packetized with appropriate addressing and size metadata before transmission through the interconnect fabric.

Following crossbar processing, the data enters the accumulation phase where it is routed to dedicated accumulator units (Accumulator A/B). These units perform critical partial sum reduction operations, combining intermediate results from multiple crossbar computations. The accumulation process ensures proper integration of distributed computations while maintaining numerical precision through careful bit-width management.

Subsequently, the accumulated data proceeds to activation units (Activation A/B) where non-linear transformations are applied. These transformations introduce the necessary non-linearity to the network while preserving the structural integrity of the data packets. The activation outputs are then prepared for spatial processing, with proper dimensional organization for downstream operations.

The final stage transitions the processed data to pooling layers through the interconnect fabric. Pooling units receive the activated feature maps and perform spatial reduction operations, systematically downsampling the data while retaining the most salient information. This generates condensed representations that are properly formatted for transmission to subsequent network layers, completing one full processing cycle from linear computation through non-linear transformation to spatial reduction.

4.2. Component Implementation

This section presents the implementation details of the key components in the proposed data traffic simulator for multi-tile CIM neural network accelerators.

All components inherit from a common `Component` base class, ensuring uniform interface implementation across the architecture. This base class enforces standardized packet handling through consistent send and receive interfaces, where components communicate exclusively through structured data packets containing source/destination addressing and payload size information. Bandwidth configuration occurs through member variables (`in_port_bw`, `out_port_bw`) that govern both intra-layer and inter-layer communication constraints. The interface design allows components to maintain internal processing state (such as the `valid_rows` and `valid_volumes` in crossbars) while exposing a consistent communication abstraction to the interconnect fabric.

4.2.1. Basic Processing Component

The foundational layer of the architecture implements primitive processing units through several classes of computational components. At the core are the matrix operation crossbars (`CIMCrossbar`), partial sum reduction accumulators (`Accumulator`), and nonlinear transformation activation units (`Activation`). These components form the basic computational building blocks that perform the fundamental mathematical operations required for neural network processing.

CIM Crossbar Component

The `CIMCrossbar` class implements a computational model of memory-resident crossbar arrays, serving as the fundamental processing unit for matrix operations in the architecture. The component's data structure tracks valid rows/columns for crossbar utilization metrics, an input counter for tracking column vectors, and configurable I/O bandwidth parameters.

The crossbar's behavior involves three key operations: (1) updating its input counter upon receiving data packets, (2) performing matrix multiplication to generate dimensionally-appropriate outputs, and (3) transmitting results through the interconnect fabric. The interconnect calculates transmission delays based on both the crossbar's bandwidth constraints (Section 3.5) and its computational model (Section 3.4.1), accurately capturing both processing and communication overhead.

Accumulator Component

The `Accumulator` class handles the accumulation of partial results from distributed crossbar computations. Its structure maintains an `input_times` counter for multi-cycle operations, tracks `compute_bits` for numerical precision management, and implements specialized bandwidth parameters (`ACC_IN_BW`, `ACC_OUT_BW`) for I/O constraints.

The accumulator's operation comprises three principal functions: (1) receiving and aggregating intermediate results from multiple crossbars, (2) scaling accumulated values by the `BIT_PRECISION` factor, and (3) transmitting reduced-size outputs through the interconnect. Delay calculations incorporate both the accumulation cycles and transmission overhead from bandwidth-constrained paths (Section 3.4.2 and Section 3.5), ensuring accurate modeling of the complete accumulation pipeline.

Activation Component

The `Activation` class implements the configuration of nonlinear transformations (ReLU, sigmoid, etc.) for neural network processing. Its structure maintains the nonlinear transformation type, `input_times` and `compute_bits` counters for operation tracking, along with specialized bandwidth parameters (`ACT_IN_BW`, `ACT_OUT_BW`) that govern data movement constraints.

The activation unit actually serves as a traffic generator: it receives incoming bits from upstream components, optionally reshapes or reorganizes them, and forwards the resulting bits to downstream processing elements. The transmission delay depends on the output data size and the bandwidth of the communication path (Section 3.4.3 and Section 3.5), capturing the dataflow latency without involving any actual computation.

4.2.2. Specialized Processing Component

Specialized processing units extend this computational foundation with domain-specific functionality. The architecture includes convolution unfolding engines (`Im2col`) that transform input tensors into column matrices suitable for crossbar operations, spatial reduction operators (`Pool`) for dimensionality re-

duction, and tensor reshaping modules (`Flatten`) that prepare multidimensional data for fully connected layers.Each specialized unit preserves the standard component interface while incorporating optimizations tailored to its layer.

Im2col Component

The `Im2col` component implements the image-to-column transformation for convolution operations. Its structure stores kernel parameters, input dimensions, and stride/padding configurations while precomputing optimal packet sizes for efficient data distribution. The transformation process involves three key steps: (1) restructuring input feature maps into column matrices(Section 2.2.3), (2) partitioning the output into fixed-size packets matching crossbar input dimensions, and (3) distributing packets across multiple crossbars through specialized multi-destination transmission. Bandwidth requirements are dynamically calculated based on the input feature map geometry and kernel parameters (Section 2.2.3 and Section 3.5), ensuring efficient resource utilization.

Pooling Component

The `Pool` component performs spatial reduction operations (max/average) for dimensionality reduction. The design maintains pooling-type configurations while tracking input bit accumulation and output packetization schemes. Each pooling operation executes three principal functions: (1) receiving activated feature maps, (2) applying kernel-based spatial reduction, and (3) generating optimally-sized output packets. The processing delay scales with input size, while transmission delay accounts for both the reduced data volume post-pooling and output path bandwidth constraints (Section 3.4.4 and Section 3.5).

Flatten Component

The `Flatten` component transforms multidimensional tensors into 1D vectors for fully-connected layers. It has a buffer accumulating all the input bits while enforcing output packet size constraints. The flattening operation proceeds through three stages: (1) receiving and concatenating multidimensional inputs, (2) restructuring data into linear format, and (3) packetizing results with size-aware handling of partial packets. The communication model employs specialized vectorized transmission that respects component size limits, with per-packet delay calculations ensuring accurate modeling of the reshaping overhead (Section 3.4.5 and Section 3.5).

4.3. Neural Network Layer Implementation

The neural network layer level serves as the mid-tier abstraction that orchestrates complete neural network layers by composing individual components into functional units. Each layer follows specific composition rules that chain components in patterns appropriate for their computational purpose.

The interconnect fabric plays a crucial role in bridging components both within and between layers. Within each layer, the first component receives inputs from either the preceding layer or external sources, while the last component establishes connections to subsequent layers. Intermediate components are precisely linked through bandwidth-configured pathways using the `Interconnect::setBandWidth()` method, ensuring proper data flow matching the layer's computational requirements. This architectural configuration enables seamless transitions across diverse layer types while maintaining uniform communication semantics.

The implementation specializes for different layer types through distinct component organization strategies. While pooling and flatten layers employ relatively simple fixed transformations, fully-connected (FCNN) and convolution (CNN) layers require sophisticated resource allocation algorithms. These complex layers must:

- Dynamically partition weight matrices across crossbar arrays;
- Configure appropriate accumulator hierarchies;
- Coordinate activation function placement;
- Establish optimized dataflow paths.

Section 4.3.1 and Section 4.3.2 detail the specific algorithms for organizing these computationally intensive layers, including crossbar allocation strategies and component interconnection patterns.

4.3.1. Fully-Connected Layer Implementation

The fully-connected layer implementation employs a systematic resource allocation strategy to map neural network parameters onto crossbar arrays. The algorithm 2, as implemented in the `FullyConnectedLayer` constructor, follows three key phases:

First, the algorithm calculates the required crossbar resources by partitioning the weight matrix. Given an input size N_{in} and neural count N_{out}, it determines the crossbar grid dimensions as:

$$N_{row} = \lceil N_{out}/V_{cb} \rceil, \quad N_{col} = \lceil N_{in}/S_{cb} \rceil \tag{4.1}$$

where $V_{cb} = S_{cb}/P$ represents the maximum vectors per crossbar, S_{cb} is the crossbar size, and P is the bit precision.

Then, the implementation instantiates crossbars in row-major order, handling edge cases for partial matrix partitions. Each crossbar's valid dimensions are set according to the remaining unallocated rows and columns, ensuring optimal resource utilization. The constructor creates corresponding accumulators and activation units for each output channel partition, maintaining a one-to-one relationship between crossbar rows and activation paths.

Finally, all components are registered with the interconnect and the bandwidth configuration, managed in `set_bandwidth()`, establishes optimized paths between components. The interconnect sets specific bandwidth constraints between:

- Crossbars and their corresponding accumulators (`CB_ACC_BW`);
- Accumulators and activation units (`ACC_ACT_BW`);
- Activation units and downstream layers (`LAYER_BW`).

Algorithm 2 Fully-Connected Layer Resource Allocation

Input: N_{in}: input size, N_{out}: output size, S_{cb}: crossbar size, B: bit precision
Output: Crossbar grid configuration with accumulators and activations
$V_{cb} \leftarrow S_{cb}/B$ {Max vectors per crossbar}
$N_{row} \leftarrow \lceil N_{out}/V_{cb} \rceil$ {Rows in crossbar grid}
$N_{col} \leftarrow \lceil N_{in}/S_{cb} \rceil$ {Columns in crossbar grid}
$R_{out} \leftarrow N_{out}$ {Remaining output neurons}
for $i \leftarrow 1$ to N_{row} **do**
  $R_{in} \leftarrow N_{in}$ {Reset remaining inputs}
  $V \leftarrow \min(V_{cb}, R_{out})$ {Current vectors}
  for $j \leftarrow 1$ to N_{col} **do**
    $S \leftarrow \min(S_{cb}, R_{in})$ {Current inputs}
    Create crossbar(S, $V \times B$) {Instantiate with valid dimensions}
    $R_{in} \leftarrow R_{in} - S$
  end for
  Create accumulator($V \times B$)
  Create activation(ACT_SIZE, type)
  $R_{out} \leftarrow R_{out} - V$
end for

The Algorithm 3 implements the forward propagation of FCNN layer and records the total delay, demonstrating efficient pipelining by:

1. Parallelizing crossbar computations within each row;
2. Synchronizing partial sums through accumulators;
3. Balancing activation outputs across multiple targets.

Algorithm 3 FCNN Forward Propagation

Input: target_address
for each input batch **do**
  $T_{total} \leftarrow 0$
  for $i \leftarrow 1$ to N_{row} **do**
    $T_{row} \leftarrow 0$
    for $j \leftarrow 1$ to N_{col} **do**
      $T_{cb} \leftarrow$ crossbar[i, j].send(accumulator[i])
      $T_{row} \leftarrow \max(T_{row}, T_{cb})$
    end for
    $T_{acc} \leftarrow$ accumulator[i].send(activation[i])
    $T_{act} \leftarrow$ activation[i].send(target_address)
    $T_{total} \leftarrow T_{total} + T_{row} + T_{acc} + T_{act}$
  end for
end for
Register all the components
Set bandwidths

4.3.2. Convolution Layer Implementation

The convolution layer implementation provides two distinct mapping strategies controlled by the `mapping_flag` parameter, illustrated in Algorithm 4.

Kernel-to-Matrix (K2M) Mapping

When `mapping_flag=true`, the implementation converts the convolution kernels to a huge matrix based on the theory discussed in Section 2.2.3. Then, the allocation method similar to Algorithm 2 is applied:

1. Calculates output feature map dimensions considering stride/padding;
2. Partitions kernels across crossbars using similar logic to FC layers;
3. Allocates accumulators per output channel;
4. Configures activation units for spatial output.

Image-to-Column (Im2Col) Mapping

When `mapping_flag=false`, the implementation employs the `Im2col` component to transform input feature maps according to the theory explained in Section 2.2.3. Key aspects include:

- Specialized bandwidth settings (`IM_CB_BW`) for Im2Col to crossbar paths;
- Dynamic input address routing based on mapping mode;
- Kernel-aware packet size optimization.

Algorithm 4 Convolution Layer Resource Allocation

Input: $I[3]$: input dimensions, $K[3]$: kernel dimensions, S: stride, P: pad
Output: Crossbar grid configuration with accumulators and activations
if mapping_flag **then**
  /* K2M mode */
  $N_{out} \leftarrow \lfloor \frac{I[0]-K[0]+2P}{S} \rfloor \times \lfloor \frac{I[1]-K[1]+2P}{S} \rfloor \times K[2]$
  $N_{in} \leftarrow I[0] \times I[1] \times I[2]$
else
  /* Im2Col mode */
  $N_{out} \leftarrow K[2]$
  $N_{in} \leftarrow K[0] \times K[1] \times I[2]$
end if
// Continue with FCNN allocation using N_{in}, N_{out}
...

The Algorithm 5 implements the forward propagation of CNN layer, handling both modes uniformly after

the initial data transformation phase and recording the delay of forward propagation. The computation follows a strict pipeline:

1. Input transformation (Im2Col when enabled);
2. Parallel crossbar operations;
3. Accumulation across spatial positions;
4. Activation and output routing.

Algorithm 5 CNN Forward Propagation

Input target_address
if not mapping_flag **then**
  $T_{im2col} \leftarrow$ im2col.send(all_crossbars)
end if
$T_{total} \leftarrow 0$
for each crossbar row i **do**
  $T_{row} \leftarrow 0$
  for each crossbar column j **do**
    $T_{cb} \leftarrow$ crossbar[i, j].send(accumulator[i])
    $T_{row} \leftarrow \max(T_{row}, T_{cb})$
  end for
  $T_{acc} \leftarrow$ accumulator[i].send(activation[i])
  $T_{act} \leftarrow$ activation[i].send(target_address)
  $T_{total} \leftarrow T_{total} + T_{row} + T_{acc} + T_{act}$
end for

Both implementations share the same bandwidth configuration strategy as FC layers, but add specialized handling for the Im2Col transformation when active.

4.4. Neural Network Implementation

The top-level neural network implementation provides a TensorFlow-style interface for constructing and executing computational graphs on the multi-tile CIM accelerator.

4.4.1. Layer Composition and Connectivity

The implementation establishes inter-layer connections through three mechanisms:

1. **Automatic Dimension Tracking**: The model maintains current tensor dimensions through `current_size[3]`, automatically calculating output shapes after each operation:

$$W_{out} = \left\lfloor \frac{W_{in} + 2P - K_w}{S} \right\rfloor + 1 \tag{4.2}$$

   where W_{in}, W_{out} are input/output widths, K_w is kernel width, S is stride, and P is padding.
2. **Interconnect Registration**: Each layer's components register with the interconnect during construction, establishing physical communication pathways:
   - Crossbars for matrix operations;
   - Accumulators for partial sums;
   - Activation units for nonlinear transforms.
3. **Dataflow Chaining**: The `forward()` method connects layers by:
   (a) Configuring first layer inputs from host;
   (b) Linking intermediate layers through `forward_propagation(get_input_addr())`;
   (c) Routing final outputs back to host.

4.4.2. Timing and Performance Analysis

The model accumulates computational delays through:

Algorithm 6 Delay Accumulation Mechanism

Input: Layer connections and bandwidth constraints
Output: Total end-to-end latency

$T_{total} \leftarrow 0$
$T_{total} \leftarrow T_{total}+$ first_layer.set_up(host)
for each layer connection i **do**
  $T_{total} \leftarrow T_{total}+$ layer[i].forward_propagation(layer[$i+1$])
end for
$T_{total} \leftarrow T_{total}+$ last_layer.forward_propagation(host)

The timing model accounts for:

- Crossbar computation latency;
- Interconnect transmission delay;
- Activation function overhead;
- Pipeline stalls from bandwidth contention.

4.4.3. TensorFlow-style Interface

The implementation provides a declarative API mirroring popular frameworks:

```
Model model({32,32,3}, 256, host, interconnect);  // Input 32x32x3
model.Conv(3,3,64).MaxPool(2,2)       // Conv->Pool
   .Conv(3,3,128).MaxPool(2,2)        // Conv->Pool
   .Flatten().Dense(256).Dense(10);   // FC layers
model.forward();                      // Execute
```

Listing 4.1: Example Network Construction

Key interface features include:

- Method chaining for concise network definition;
- Automatic dimension inference;
- Default parameters (stride=1, pad=0, ReLU);
- Polymorphic layer handling.

The implementation efficiently manages resources through:

- Dynamic crossbar allocation based on layer requirements;
- Automatic bandwidth configuration between components;
- Memory-aware tensor reshaping.

4.5. Interconnect Implementation

The interconnect fabric serves as the central communication backbone of the multi-tile CIM accelerator, implementing a three-tiered architecture that connects components, layers, and complete neural networks. As illustrated in Figure 4.1, this critical subsystem manages data movement while simultaneously collecting performance metrics.

4.5.1. Data Structure Organization

The interconnect class maintains several key data structures that enable its functionality:

- `address_map`: A bi-directional registry mapping addresses to component instances;
- `bandwidth_map`: A sparse matrix storing configured bandwidth constraints between components;

- `logger`: logs the data traffic and the log can be used to generate annotated topology graphs through Graphviz [10], which will be detailed in Section 4.6;
- `total_bits_transferred`: Cumulative counter of all data movement;
- `crossbar_num & crossbar_valid_area`: Records computation resource usage statistics.

These elements work in concert to provide both real-time communication services and post-simulation analysis capabilities.

4.5.2. Address Space Management

The interconnect implements a hierarchical addressing scheme through Algorithm 7, which guarantees unique component identification across all abstraction levels.

Algorithm 7 Component Registration Protocol

Input: Component pointer *comp*, size parameter *size*
Output: Assigned hardware address *addr*

addr $\leftarrow$ *next_addr* {Current address counter}
address_map[*addr*] $\leftarrow$ *comp* {Register mapping}
next_addr $\leftarrow$ *next_addr* $+$ *UNIT_ADDR* {Increment counter}
if *comp* is CIMCrossbar **then**
  crossbar_num $\leftarrow$ *crossbar_num* $+ 1$
  valid_area $\leftarrow$ *valid_area* $+$ *comp*.getValidArea()
end if
return *addr*

Key properties of the addressing scheme include:

- **Uniform addressing**: Consistent namespace from components to complete networks;
- **Strided allocation**: *UNIT_ADDR* intervals prevent collisions;
- **Resource tracking**: Special handling for computation-in-memory crossbars.

4.5.3. Bandwidth-Constrained Routing

The interconnect's routing engine implements precise delay calculations through the bandwidth-constrained communication model(Section 3.5):

$$t_{\text{transmit}} = \left\lceil \frac{\text{size\_bits}}{\min(bw_{\text{src}}, bw_{\text{dst}}, bw_{\text{path}})} \right\rceil \times t_{\text{cycle}} \tag{4.3}$$

Algorithm 8 details the complete transmission process:

Algorithm 8 Packet Transmission Procedure

Input: Source address *src*, destination address *dst*, packet size *size_bits*
Output: Transmission delay *delay*

$bw_{\text{src}} \leftarrow$ src.getOutPortBW() {Source bandwidth}
$bw_{\text{dst}} \leftarrow$ dst.getInPortBW() {Destination bandwidth}
$bw_{\text{path}} \leftarrow bandwidth\_map[(\text{src}, \text{dst})]$ {Path constraint}
$bw_{\text{eff}} \leftarrow \min(bw_{\text{src}}, bw_{\text{dst}}, bw_{\text{path}})$
delay $\leftarrow \lceil$ *size_bits*$/bw_{\text{eff}} \rceil \times$ *UNIT_TIME*
total_bits $\leftarrow$ *total_bits* $+$ *size_bits* {Update metrics}
min_bw $\leftarrow \min($*min_bw*$, bw_{\text{eff}})$
logger.recordTransmission(src, dst, *size_bits*)
return *delay*

4.5.4. Performance Monitoring Framework

The interconnect collects essential performance metrics through the monitoring framework, which is described in Table 4.1.

Table 4.1: Interconnect Monitoring Metrics

Metric	Description
$\mathcal{C}$	Crossbar resources (count and utilization percentage)
$\mathcal{T}_{\text{bits}}$	Total data volume transferred (bits)
$\mathcal{D}_{\text{e2e}}$	End-to-end latency of complete forward passes

This comprehensive interconnect architecture enables accurate modeling of both computational and communication aspects in multi-tile CIM accelerators, providing critical insights for architecture optimization and performance tuning. The implementation faithfully captures the trade-offs between computation efficiency and communication constraints that characterize real-world NN accelerators.

4.6. Topology Visualization

The system includes a topology visualization feature that captures the connectivity and data flow between components in real time. This functionality is integrated directly into the `Interconnect` subsystem and leverages *Graphviz* to generate clear and comprehensive diagrams of the component network.

4.6.1. Implementation

The visualization is realized through the `DotGraphLogger` class, which provides the following capabilities:

- **Graph Initialization**: Creates and manages Graphviz DOT file output;
- **Node Management**: Tracks system components with formatted labels including addresses and types;
- **Edge Creation**: Records data transfers between components with detailed metadata (size and frequency);
- **Automatic Finalization**: Ensures correct file closure and DOT formatting.

The logger is seamlessly integrated into the `Interconnect` class:

```
class Interconnect {
private:
    DotGraphLogger logger;  // Visualization logger
    // ...

public:
    Interconnect(const std::string& dotFileName); // Constructor
    // ...
};
```

Listing 4.2: Interconnect Class Integration

4.6.2. Key Features

The visualization functionality introduces three main features: real-time data capture, comprehensive metadata, and implicit node management.

Real-time Data Capture

Component interactions are captured during runtime as directed edges between nodes. For example:

```
uint32_t Interconnect::sendPacket(const Packet& packet) {
    // ... processing logic
    logger.addEdge(packet.source,
                   address_map.find(packet.source)->second->getType(),
                   packet.destination,
                   address_map.find(packet.destination)->second->getType(),
                   packet.size_bits, 1);
    // ...
}
```

Listing 4.3: Real-time Data Logging

Comprehensive Edge Metadata

Each edge in the generated DOT graph includes:

- **Source and destination components**, labeled with addresses and types;
- **Data transfer size**, reported in bits;
- **Transfer frequency**, indicating the number of occurrences.

Implicit Node Management

Nodes are automatically created upon edge insertion, ensuring that the graph only contains components that actually participate in data transfers:

```
void DotGraphLogger::addEdge(uint32_t from, const std::string& fromType,
                             uint32_t to, const std::string& toType,
                             uint32_t sizeBits, uint32_t times) {
    std::string fromNode = formatNode(from, fromType);
    std::string toNode = formatNode(to, toType);

    dotFile << "  \"" << fromNode << "\" -> \"" << toNode
            << "\" [label=\"" << std::dec << times << "x "
            << std::dec << sizeBits << " bits\"];\n";
}
```

Listing 4.4: Edge-based Node Creation

4.6.3. Usage Example

The following example creates a simple data flow pipeline where:

- A `Host` sends 256 bits of data to a `CIMCrossbar`;
- The `CIMCrossbar` processes the data (valid column width: 64) and sends 64 bits to an `Accumulator`;
- Bandwidth constraints are defined between components (1,000 bits/unit time for host-to-crossbar, 800 bits/unit time for crossbar-to-accumulator);
- All transfers are automatically logged into `topology.dot`.

```
// Create interconnect with visualization
Interconnect ic("topology.dot");

// Register components
CIMCrossbar crossbar(CROSSBAR_SIZE, &ic, 256, 64);
Host host(1024, &ic);
Accumulator acc(512, &ic);

// Set bandwidth constraints
ic.setBandWidth(host.getAddress(), crossbar.getAddress(), 1000);
ic.setBandWidth(crossbar.getAddress(), acc.getAddress(), 800);

// Data transfers are logged automatically
host.send(crossbar.getAddress(), 256);
crossbar.send(acc.getAddress());
```

Listing 4.5: System Usage Example

The generated DOT file is shown below:

```
digraph InterconnectGraph {
  "0x1000 Host" -> "0x2000 Crossbar" [label="1x 256 bits"];
  "0x2000 Crossbar" -> "0x3000 Accumulator" [label="1x 64 bits"];
}
```

Listing 4.6: DOT File Output Format

This DOT description can be rendered with Graphviz, e.g., `dot -Tpng topology.dot -o topology.png`, producing the visualization in Figure 4.3. The diagram illustrates that the Host (0x1000) sends 256 bits to the Crossbar (0x2000) once, and the Crossbar subsequently transmits 64 bits to the Accumulator (0x3000). The edge labels in the diagram show frequency and size of exchanged data.

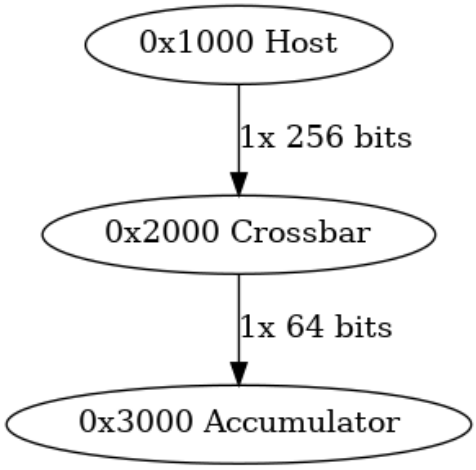


Figure 4.3: Visualization of connectivity and data flow from the example.

4.6.4. Benefits

The topology visualization provides multiple benefits:

1. **Debugging Aid**: Simplifies detection of connectivity or configuration issues;
2. **Performance Analysis**: Reveals data flow patterns and possible communication bottlenecks;
3. **Documentation**: Generators up-to-date architecture diagrams;

Overall, this feature provides a non-intrusive and effective mechanism for visualizing and analyzing the complex interactions and bit-level data exchanges between components in the architecture.

4.7. Conclusion

This chapter presented a comprehensive implementation of the proposed multi-tile CIM accelerator simulator, detailing its three-level hierarchical architecture from fundamental components to complete neural networks. The implementation accurately models both computational operations and communication constraints through a bandwidth-aware interconnect fabric that functions as the central nervous system of the accelerator. By maintaining consistent interfaces across all abstraction levels and integrating detailed performance monitoring capabilities, the simulator offers a robust framework for analyzing the complex trade-offs between computational efficiency and communication overhead in realistic neural network workloads. The inclusion of topology visualization further strengthens its analytical utility by providing intuitive insights into data flow patterns and potential architectural bottlenecks. Overall, the chapter delivers a practical and extensible simulation platform that bridges device-level modeling with system-level evaluation, enabling comprehensive exploration of CIM accelerator design.

5

Results and Discussion

This chapter presents experimental results and analysis of FCNN and CNN implementations on MNIST using the proposed data traffic simulator. Section 5.1 gives an overview of the Neural Network architectures for the experiment. Section 5.2 presents the topology graph of the FCNN to demonstrate the framework's visualization capabilities. Section 5.3 details the evaluation methodology for key metrics including crossbar utilization, total transferred bits, crossbar amount and traffic delay. Section 5.4 analyzes the results of FCNN, exploring how crossbar size, bit precision and bandwidth affect the metrices. Section 5.5 analyzes the results of CNN, and mainly compares Im2Col and K2M conversion methods, revealing fundamental trade-offs between computational efficiency and resource requirements, culminating in practical design guidelines for different application constraints. Section 5.6 summarizes the experimental results.

5.1. Neural Network Architectures

The study employed two distinct neural network architectures for processing individual images from the MNIST dataset. Both architectures take a single 28×28 grayscale image as input. The specifications of each architecture are detailed below.

5.1.1. Fully Connected Neural Network Architecture

The FCNN architecture processes flattened image data through multiple dense layers with ReLU activation functions, culminating in a softmax output layer for 10-class classification. The layer-by-layer specification of the proposed FCNN is presented in Table 5.1. Each row corresponds to a layer in the model, while the columns capture different descriptive attributes:

- **Layer Type** specifies the role of the layer in the network (e.g., input, hidden dense layers, output).
- **Output Shape** indicates the dimensionality of the data as it exits the layer. The format follows the convention (n), where n is the number of elements in the output vector. For example, the input layer produces a vector of size 784, while the first hidden layer outputs 512 activations.
- **Number of Units** denotes the number of neurons (trainable units) in the layer. This field is only applicable to dense layers, so it is marked with "–" for the input layer.
- **Activation** specifies the activation function applied at the layer's output. For hidden layers, ReLU is used, while the output layer applies the softmax function. Layers without a non-linear transformation ,such as the input layer, are marked with "–".

Table 5.1: Architecture of the Fully Connected Neural Network (FCNN) for MNIST

Layer Type	Output Shape	Number of Units	Activation
Input Layer	(784)	–	–
Dense (Hidden Layer 1)	(512)	512	ReLU
Dense (Hidden Layer 2)	(32)	32	ReLU
Dense (Output Layer)	(10)	10	Softmax

5.1.2. Convolution Neural Network Architecture

The proposed CNN architecture processes individual images through convolution and pooling layers for feature extraction, followed by dense layers for classification. The layer-by-layer specification of the proposed CNN is shown in Table 5.2. Each row corresponds to a layer in the model, and the columns capture the following descriptive attributes:

- **Layer Type:** Specifies the type of layer, such as convolution, pooling, flatten, or dense.
- **Kernel/Stride:** Indicates the size of the convolution or pooling kernel and the stride applied. For layers that do not use a kernel (e.g., input, flatten, dense), this field is marked with "–".
- **Output Shape:** Gives the dimensionality of the data as it exits the layer. For convolutional and pooling layers, the format (H, W, C) denotes the height, width, and number of channels of the output feature map. For flatten and dense layers, the convention (n) is used, the same as in Table 5.1, where n is the dimension of the output vector.
- **Parameters:** Reports the number of trainable parameters (weights and biases) in the layer. Layers without parameters (e.g., input, pooling, flatten) are marked with "0".
- **Activation:** Same convention as in Table 5.1: convolutional and hidden dense layers use ReLU, the output dense layer uses softmax, and layers without an activation function are marked with "–".

Table 5.2: Architecture of the Proposed Convolution Neural Network for MNIST

Layer Type	Kernel/Stride	Output Shape	Parameters	Activation
Input	–	(28, 28, 1)	0	–
Conv2D	3×3, stride 1	(28, 28, 32)	320	ReLU
MaxPool2D	2×2, stride 2	(14, 14, 32)	0	–
Conv2D	3×3, stride 1	(14, 14, 64)	18,496	ReLU
MaxPool2D	2×2, stride 2	(7, 7, 64)	0	–
Conv2D	3×3, stride 1	(7, 7, 64)	36,928	ReLU
Flatten	–	(3136)	0	–
Dense	–	(64)	200,768	ReLU
Dense	–	(10)	650	Softmax

5.2. Topology Visualization Case of FCNN

To demonstrate the framework's architectural visualization capabilities, Figure 5.1 presents the generated topology for the given FCNN implemented using 512×512 crossbars with 2-bit precision. The topology reveals three critical aspects of the architecture:

- **Weight Matrix Partitioning**:
  - Explicit mapping of sliced weight matrices to crossbar arrays (0x2000-0x5000);
  - Visual confirmation of tiling patterns matching the neural network's layers;
  - Clear display of matrix slicing granularity (512$\times$512 blocks).
- **Precision-Aware Dataflow**:
  - 2-bit parallel datapaths;
  - Progressive bit-width reduction following network architecture;
  - Precision-preserving transitions between components with explicit annotation of the number of bits.
- **Computational Pipeline**:
  - Crossbar arrays for matrix-vector multiplication;
  - Paired accumulators and activation units;

The visualization enables verification of:

1. Correct weight matrix partitioning and crossbar mapping;
2. Consistency between expected and visualized bit-level data traffic across interconnections;

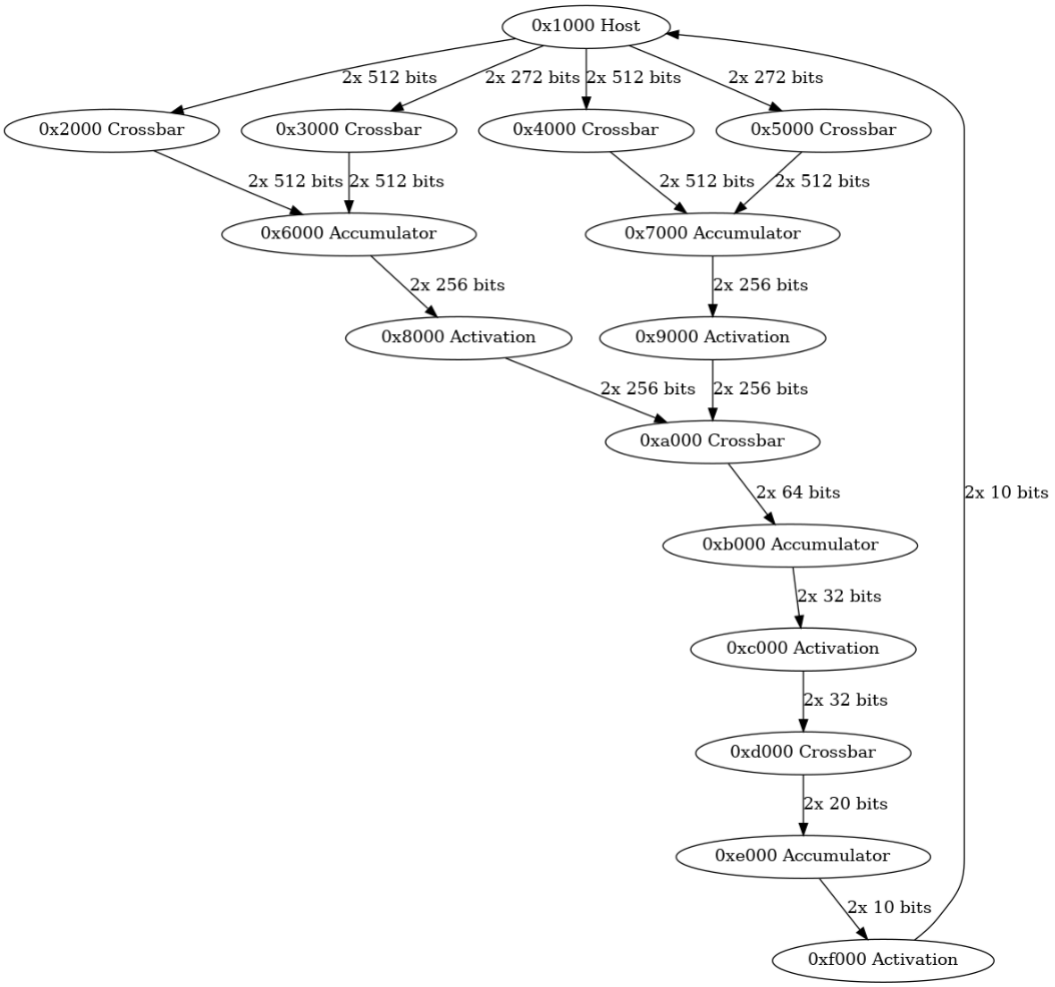


Figure 5.1: Architecture topology of the FCNN visualization.

3. Preservation of 2-bit precision through all transformations;
4. Proper accumulator-activation pairing relationships;
5. Dataflow continuity across sliced weight operations.

Particularly valuable is the explicit confirmation that the $784 \rightarrow 512 \rightarrow 32 \rightarrow 10$ network's weight matrices are properly sliced and distributed across the crossbar arrays while maintaining correct dimensional reduction and 2-bit precision constraints throughout all parallel datapaths. Additionally, the visualization exposes the amount of bits exchanged between crossbars, accumulators, and activation units, making bandwidth requirements and potential communication bottlenecks directly observable alongside the structural and precision-related aspects of the architecture.

5.3. Analysis Methodology

According to Section 4.5, the recorded metrics include:

- Crossbar utilization (U);
- Total transferred bits (B_{total});
- Number of crossbars (N_{xbar});
- End-to-end clock delay (D_{e2e}).

For a fixed neural network (NN) structure, the input variables are:

- Crossbar size (S);
- Bit precision (P);

- Bandwidth (W).

5.3.1. Analysis of Static Metrics

Based on the bandwidth model in Section 3.5, only D_{e2e} is affected by W. Therefore, for the first three metrics (U, B_{total}, N_{xbar}), the manuscript analyzes their relationship with S using dot-line charts, with each line representing a fixed P. Thus, for a given P, each S maps to specific values of these metrics:

$$\begin{cases} U = f_1(S, P) \\ B_{total} = f_2(S, P) \\ N_{xbar} = f_3(S, P) \end{cases} \tag{5.1}$$

5.3.2. Delay Analysis

For D_{e2e}, we first visualize the relationships through heat maps where:

- X-axis: S;
- Y-axis: W;
- Each map corresponds to a specific P;
- Cell color intensity represents D_{e2e}.

Next, for each fixed P, we identify the optimal crossbar size S^*_{delay} that minimizes D_{e2e} for a given W:

$$S^*_{delay} = \arg\min_{S} D_{e2e}(S, W, P) \tag{5.2}$$

Consequently, for fixed P, each W determines:

- The optimal crossbar size S^*_{delay};
- Corresponding values of U, B_{total}, and N_{xbar} through Eq. 5.1.

5.4. Results and Analysis of FCNN

The experimental setup employs the following configuration parameters:

- Crossbar size: $S = 2^n \times 2^n$ where $n \in [5, 10]$;
- Bit precision: 1-bit, 4-bit, and 8-bit representations;
- Uniform bandwidth across all ports and channels, varying from 16 to 1024 bits/clock cycle.

5.4.1. Static Metrics Analysis

Figure 5.2 presents three key static metrics—crossbar utilization, total transferred bits, and crossbar count—evaluated across different crossbar sizes and bit precisions. In each chart, the x-axis denotes the crossbar size, the y-axis denotes the corresponding metric, and each line corresponds to a specific bit precision.

Crossbar Utilization exhibits an inverse relationship with crossbar size, decreasing from 0.95 to less than 0.1 for 1-bit precision and from 0.95 to 0.55 for 8-bit precision as S increases. This reduction stems from increased fragmentation in larger crossbars. Notably, the utilization decrease slows between $2^8 \times 2^8$ and $2^9 \times 2^9$ for 4-bit and 8-bit precisions because the fixed 784-element input vector results in the same first-layer utilization of $784/1,024$ for both crossbar sizes. The subsequent layers are too small to significantly affect the overall utilization. This effect is less pronounced at smaller crossbar sizes, where the influence of the second and third layers becomes more noticeable.

Total Transferred Bits demonstrates positive correlation with bit precision and negative correlation with crossbar size. Larger crossbars reduce the required data transfer by decreasing the number of column accumulations needed for weight matrix operations.

Crossbar Count follows similar trends to transferred bits, increasing with higher precision requirements and decreasing with larger crossbar sizes.

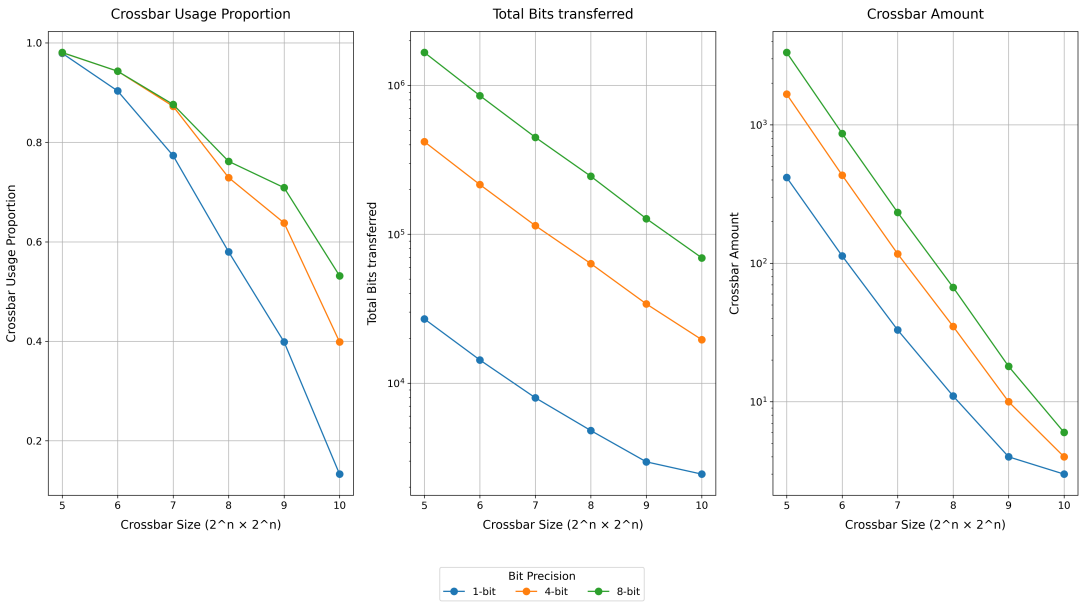


Figure 5.2: Static metrics: crossbar utilization, total transferred bits, and crossbar count.

5.4.2. Crossbar Size Optimization for Bandwidth

The optimization process for 1-bit precision configurations yields the optimal crossbar sizes:

$$S^*_{delay}(W) = \arg\min_{S} D_{e2e}(S, W, P = 1\text{-bit}) \tag{5.3}$$

Figure 5.3 illustrates the trade-off between crossbar size and delay across a range of bandwidth configurations (from 16 to 1,024 bits). The stars mark the optimal crossbar sizes for each bandwidth, where the smallest crossbar achieves the lowest delay, while the optimal frontier curve connects these Pareto-efficient points. Additionally, two significant bandwidth thresholds emerge from this analysis:

- **Critical Bandwidth** ($W_{crit} = 256$): Below this threshold, the delay first decreases, then increases, and eventually stabilizes. Beyond the threshold, the delay decreases monotonically with crossbar sizes;
- **Saturation Bandwidth** ($W_{sat} = 512$): No further delay improvement occurs when bandwidth exceeds this value.

In general, the optimal crossbar size S^*_{delay} generally increases with increasing available bandwidth, suggesting a trade-off between these parameters.

5.4.3. Multi-Precision Bandwidth Analysis

Extending the optimization of Section 5.4.2 to multi-precision configurations leads to Figure 5.4, where the x-axis denotes bandwidth(bits per unit time), the y-axises denote lowest delay and optimal crossbar size, and each line corresponds to a specific bit precision. Several key insights are revealed:

- The minimal achievable delay shows diminishing returns with increasing bandwidth, reaching a stable plateau after initial rapid improvement;
- Lower bit precision generally results in shorter delay, though the initial decrease is more gradual;
- Optimal crossbar sizes generally increase with available bandwidth, though the relationship is non-monotonic;
- The optimal crossbar size is the same for 4-bit and 8-bit precision. For 1-bit precision, the optimal size is generally not larger than those of the other two, except when limited by a 512-bit bandwidth, which results in a larger optimal crossbar size.

The inflection point in the delay-bandwidth relationship represents a particularly efficient operating point, simultaneously minimizing delay while requiring relatively modest crossbar sizes. This observation holds across all tested precision configurations.

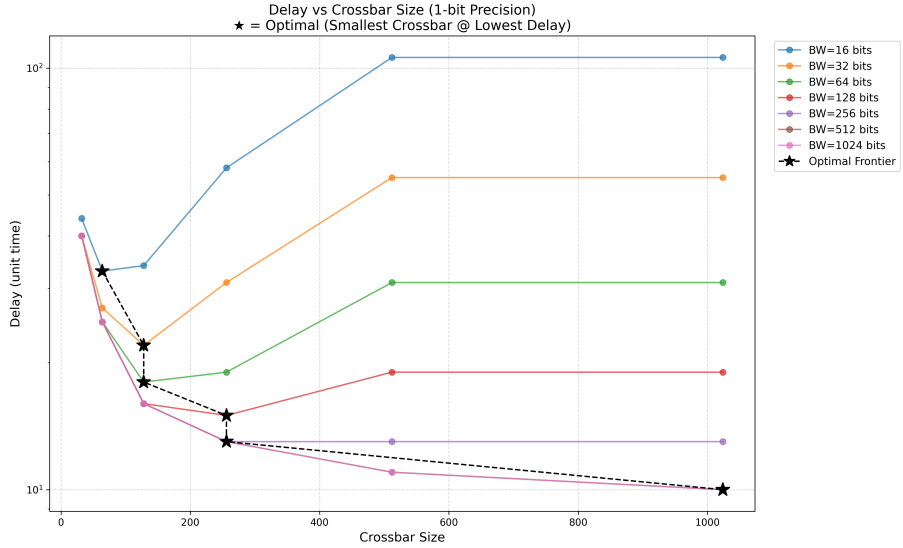


Figure 5.3: Delay optimization analysis for 1-bit precision configurations.

5.5. Results and Analysis of CNN

The CNN configurations maintain identical parameters to the FCNN implementation described in Section 5.4. Given the presence of convolution layers, both Im2Col (Section 2.2.3) and K2M (Section 2.2.3) conversions are evaluated, with particular emphasis on their comparative performance metrics.

5.5.1. Static Metrics Analysis

The same as Figure 5.2, Figure 5.5 presents three fundamental static metrics evaluated across different crossbar sizes and bit precisions..

Crossbar Utilization though generally decreases with crossbar size, demonstrates distinct patterns between conversion methods. K2M maintains consistently high utilization (0.8-1.0) across all crossbar sizes, decreasing only 20% or so as size increases from $2^5 \times 2^5$ to $2^{10} \times 2^{10}$. In contrast, Im2Col shows rapid utilization decay from almost 1.0 to less than 0.1 over the same range. This divergence stems from K2M's generation of large, densely packed weight matrices that fully utilize crossbar resources, while Im2Col's smaller matrices lead to fragmentation, particularly in marginal computation batches. Additionally, lower bit precision results in reduced utilization for both methods, though the difference between 4-bit and 8-bit precision remains minor.

Total Bits Transferred reveals that Im2Col requires much fewer bits than K2M across all configurations, because K2M's expanded matrix representations of convolution kernels. Both methods show negative correlation with crossbar size, though there exists saturation for Im2Col, where additional size provides nearly no reduction. In addition, the total bits of both methods increase constantly with higher bit precision.

Crossbar Count K2M requires significantly higher resource counts (10^2–10^6), which consistently decrease as crossbar size increases. In contrast, Im2Col exhibits a more moderate decline, particularly for crossbars larger than $2^6 \times 2^6$. Regarding bit precision, the crossbar count for K2M scales positively with the bit precision, whereas Im2Col's curves gradually converge, indicating better scalability for high-precision implementations.

5.5.2. Multi-Precision Bandwidth Analysis

Extending the bandwidth optimization methodology from Section 5.4.2 yields the results shown in Figures 5.6. Similar to Figure 5.4, Figures 5.6a and 5.6b present the minimal achievable delays and the corresponding optimal crossbar sizes across different bandwidths for the K2M and Im2Col methods, where the x-axis denotes bandwidth(bits per unit time), the y-axises denote lowest delay and optimal crossbar size, and each line corresponds to a specific bit precision. Several key insights are revealed:

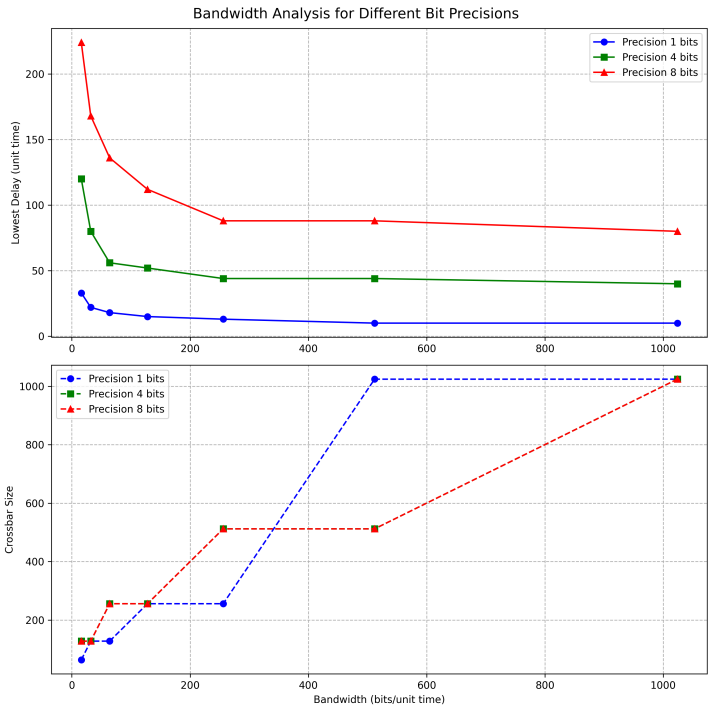


Figure 5.4: Multi-precision bandwidth analysis: delays (top) and crossbar sizes (bottom).

- Both methods gain lower latency with larger bandwidth or lower bit precision;
- Both methods generally exhibit monotonic optimal crossbar size selection;
- K2M demonstrates superior latency characteristics, maintaining delays 2 orders of magnitude lower than Im2Col across all bandwidth constraints (10^1-10^3 cycles vs 10^3-10^5 cycles);
- Im2Col's delay profile mirrors FCNN behavior, with consistent 256×256 optimal crossbar size for 1-bit precision regardless of bandwidth.

5.5.3. Comparative Analysis of Conversion Methods

The evaluation reveals fundamental trade-offs between K2M and Im2Col approaches:

K2M Advantages

- **Computational Efficiency**: 100× lower latency and maximal crossbar utilization;
- **Precision Scalability**: Consistent performance gains across 1-8 bit precision levels.

Im2Col Advantages

- **Resource Efficiency**: 10-1,000× reduction in crossbar count requirements;
- **Energy Efficiency**: mostly 10× reduction in data transfers.

Design Guidelines The choice between methods depends on application constraints:

- **Latency-Critical Systems**: K2M is preferable when area/power budgets permit;
- **Area/Energy-Constrained Designs**: Im2Col offers superior efficiency.

This analysis demonstrates that while K2M achieves superior computational performance through aggressive resource utilization, Im2Col remains competitive for resource-constrained implementations. Furthermore, future architectures may benefit from hybrid approaches that dynamically select conversion methods based on layer-specific requirements.

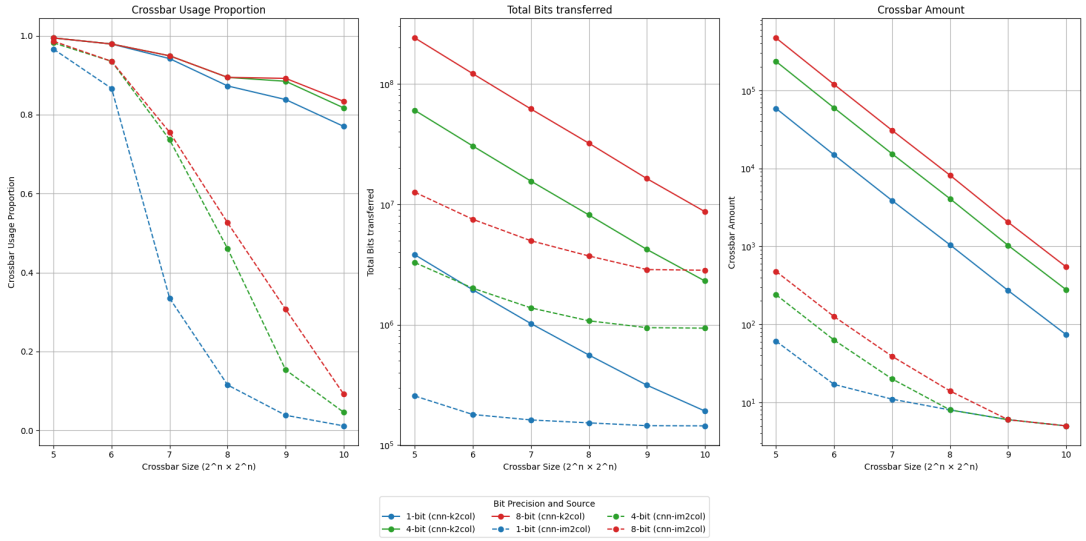
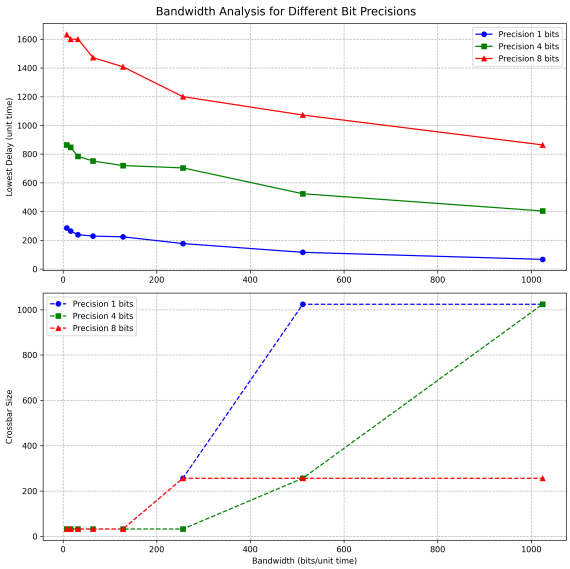


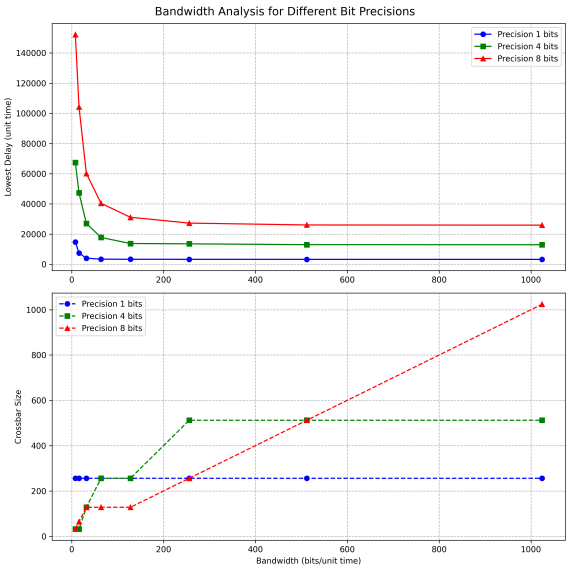
Figure 5.5: Crossbar utilization, data transfer, and count across sizes and precisions (K2M vs. Im2Col).

5.6. Conclusion

This chapter presented and analyzed the experimental results of FCNN and CNN implementations on MNIST using the proposed data traffic simulator. For FCNNs, the results showed that crossbar utilization decreases with larger crossbar sizes, while total transferred bits and crossbar counts increase with higher bit precision but improve with larger crossbars. Delay analysis revealed two critical thresholds: a *critical bandwidth* of 256 bits/clock, below which delay fluctuates, and a *saturation bandwidth* of 512 bits/clock, beyond which no further delay improvement is observed. For CNNs, comparison of K2M and Im2Col methods highlighted clear contrasts: K2M achieved up to 100× lower latency and consistently high utilization, but required up to 10^6 crossbars and far greater data movement; in contrast, Im2Col reduced crossbar count by up to three orders of magnitude and cut transferred bits by about 10×, though at the cost of longer delays (10^3–10^5 cycles). Together, these findings underline fundamental trade-offs between latency and efficiency, providing concrete design guidelines—K2M for performance-critical systems and Im2Col for area- or energy-constrained designs.



(a) K2M: (Top) minimal achievable delays and (Bottom) corresponding optimal crossbar sizes.



(b) Im2Col: (Top) minimal achievable delays and (Bottom) corresponding optimal crossbar sizes.

Figure 5.6: Multi-precision bandwidth analysis of (a) K2M and (b) Im2Col.

6

Conclusions

This chapter summarizes all the works of the manuscript and provides some potential future research directions. Section 6.1 provides a summary of the thesis. Section 6.2 offers some promising directions.

6.1. Summary

Chapter 1 established the research foundation by identifying key limitations in von Neumann architectures and formulating specific gaps in current multi-tile CIM design methodologies. Therefore, the thesis focused on filling the gaps in certain application scenarios, developing a network traffic simulation framework for multi-tile CIM accelerator in NN processing.

Chapter 2 established the theoretical foundations of CIM architectures. For neural network implementations, it identified critical challenges in matrix-to-crossbar mapping and dataflow optimization, particularly for convolution layers which require specialized vector-matrix conversion. Two conversion methods were analyzed: Im2Col, which enables dense computation at the cost of input duplication, and K2M, which preserves input structure but produces large sparse matrices.

Chapter 3 presented a systematic modeling framework that abstracts neural network operations into hardware-executable patterns. It detailed MVM operations in resistive crossbars with mapping and tiling strategies, introduced top-down modeling from Neural Network level to component level, and incorporated a bandwidth-constrained communication model to enhance the model's realism.

Chapter 4 described the hierarchical simulator, from component-level modules to full neural network integration, based on the modeling framework. It also implemented an interconnect subsystem, which manages the connection between components and records metrics to quantify computational resource utilization and communication efficiency. Furthermore, the chapter introduced a topology visualization functionality integrated into the interconnect subsystem, which captures connectivity and dataflow during simulation and generates Graphviz-based diagrams.

In general, the work made three key contributions:

- **Parameterized optimization framework**: the simulation framework accepts high-level specifications such as network architecture, bit precision, crossbar size, and bandwidth constraints, and automatically generates hardware mappings through efficient matrix-to-crossbar allocation and interconnect topology construction. This provides a structured methodology for design-space exploration, integrating both algorithmic and hardware considerations.
- **Quantitative evaluation framework**: the interconnect subsystem records metrics such as average crossbar utilization (resource efficiency), total crossbar count (hardware cost), bit-transfer counts (energy proxy), and total inference cycles (latency), enabling multidimensional assessment of multi-tile CIM accelerator performance, offering deeper insight into performance bottlenecks and guiding more informed design optimizations.
- **Topology visualization**: component interconnections recorded by the interconnect subsystem are visualized using Graphviz. The intuitive diagram allows rapid inspection of connectivity and

dataflow, accelerating early-stage design by helping designers detect mismatches, identify bottlenecks, and make informed optimization decisions before detailed implementation.

Finally, Chapter 5 validated the framework through a comprehensive evaluation of FCNN and CNN implementations on MNIST using the proposed CIM simulator, analyzing four key metrics across varying crossbar sizes, bit precisions, and bandwidth constraints. For FCNNs, results showed a fundamental trade-off between crossbar utilization and computational latency. CNN analysis highlighted differences between Im2Col and K2M: K2M achieved up to 100× lower latency through maximal utilization (0.8–1.0), while Im2Col reduced crossbar count and data transfer energy by 10–1,000×. These results suggested design guidelines: K2M suits latency-critical systems with sufficient resources, whereas Im2Col favors area- or energy-constrained applications, pointing to potential hybrid approaches that select methods per layer. Experiments also revealed the effect of bandwidth on optimal crossbar size: systems with higher bandwidth capacity usually benefit from larger crossbar arrays, achieving lower delays, but at the cost of reduced utilization.

Reproducibility The complete source code of the multi-tile CIM traffic simulation framework is publicly available at https://github.com/Zz-sev-point/IC_SIM. This repository provides instructions, example configurations, and scripts to reproduce the experiments presented in this thesis.

6.2. Future Work

This section outlines several promising directions to enhance the proposed traffic simulation framework for CIM accelerator:

- **Mapping Optimization**: While the current framework provides effective models for mapping MVM operations in FCNN and convolution operations in CNN to CIM crossbars, further optimization of these mapping algorithms could yield significant improvements in resource utilization and reduction of data traffic overhead. Potential approaches include dynamic tile-size adaptation, optimizing spatial or temporal chunking of feature maps and weights to minimize data transfers and maximize hardware utilization, as explored in recent mapping strategies for CIM workloads [21]. Additionally, intelligent bit-slicing strategies for multi-bit precision operations can both improve precision and reduce resource usage. For instance, improved bit-level mapping schemes reconfigure weight storage within crossbar arrays to minimize array count, while bit-slicing techniques compensate for analog variability in memory elements and reconstruct high-precision results via partial outputs [5, 34]. Beyond tiling and bit-slicing, sparsity-adaptive mapping methods can further enhance utilization by dynamically packing non-zero weights and avoiding idle devices, as demonstrated in sparsity-aware CIM accelerators [**Zhang_2024**, 18].
- **Heterogeneous Configuration Support**: The framework could be extended to support heterogeneous crossbar configurations, where:
  - Different crossbar sizes are employed for different NN layers to maximize resource utilization;
  - Variable bandwidth allocation across ports and channels is able to better reflects real-world hardware constraints.

  The existing framework architecture already provides interfaces for such multi-configuration simulations, requiring primarily implementation extensions.
- **Hybrid Conversion Methodology**: Current limitations require selecting either Im2Col or K2M conversion methods for entire simulations. Future development should enable simultaneous simulation of heterogeneous systems containing multiple accelerator cores optimized for different conversion approaches. This would better model practical systems that may employ specialized cores for different layer types or computational patterns.
- **Conflict-Aware Delay Modeling**: As identified in Section 3.5, the current bandwidth model does not account for access conflicts in shared-resource scenarios. Future developments will focus on modeling contention effects when multiple crossbars simultaneously access accumulators. This will involve implementing configurable arbitration protocols (e.g., time-multiplexed, priority-based, or hardware-managed) with associated control-overhead modeling, drawing on designs and analyses of arbitration schemes in digital systems [9]. Queuing theory will be integrated to estimate contention-induced delays under varying traffic patterns, while preserving the framework's cycle-

accurate simulation capability. These enhancements will enable more realistic performance prediction for architectures employing resource-sharing optimization techniques.

- **Support for Diverse Neural Network Layers**: The framework will be extended to support contemporary neural network architectures beyond basic fully-connected and convolution layers. Priority additions include:
  - **Recurrent layers (LSTM, GRU)**: Real-hardware implementations of recurrent models using CIM have been demonstrated—for instance, LSTM networks realized in memristor crossbar arrays, enabling high-density, low-latency edge inference [25].
  - **Attention layers for transformers**: Attention mechanisms, a core component of transformer architectures, have been incorporated into CIM accelerators. Designs such as Attar (RRAM-based attention engine with in-memory softmax) and analog in-memory attention architectures highlight practical crossbar-based implementations of Q/K/V computations and softmax [24, 22].
  - **Sparse convolution layers**: Sparse attention accelerators like CPSAA utilize crossbar-based PIM to process sparse attention patterns efficiently, which can be generalized to sparse convolution kernels as well [**Li2022CPSAA**].

  Each layer type will implement its unique crossbar mapping strategy and precision requirements. This expansion will significantly increase the framework's practical utility for tasks like neural architecture search and accelerator co-design.
- **Training Operation Simulation**: While currently limited to forward propagation, the framework will be extended to model the complete training cycle, including both forward and backward propagation [36]. This involves:
  1. Gradient computation through crossbar arrays with non-ideal conductance updates, as modeled by simulation tools like *TxSim*, which captures forward, backward, and weight-update non-idealities in resistive crossbars [35];
  2. Weight update noise modeling accounting for device programming variability;
  3. Precision management during backpropagation operations.

  Special attention will be given to in-situ training algorithms that leverage CIM's parallel computation capabilities. The simulation will track training-specific metrics such as convergence rate under precision constraints and gradient quality degradation. This capability will open new research directions in on-device learning and adaptive edge AI systems.

References

[1] Baeldung. *Convolution as Matrix Multiplication*. https://www.baeldung.com/cs/convolution-matrix-multiplication. Accessed: 2023-11-01. 2023.

[2] Rajendra Bishnoi et al. "Energy-efficient Computation-In-Memory Architecture using Emerging Technologies". In: *2023 International Conference on Microelectronics (ICM)*. 2023, pp. 325–334. DOI: 10.1109/ICM60448.2023.10378889.

[3] Tiancheng Cao et al. "Parasitic-Aware Modelling for Neural Networks Implemented with Memristor Crossbar Array". In: *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. 2021, pp. 122–126. DOI: 10.1109/MCSoC51149.2021.00025.

[4] Xiang Chen et al. *Edge Computing Enabled Real-Time Video Analysis via Adaptive Spatial-Temporal Semantic Filtering*. 2024. arXiv: 2402.18927 [cs.CV]. URL: https://arxiv.org/abs/2402.18927.

[5] Zhenjiao Chen et al. "Advancing Mapping Strategies and Circuit Optimization for Signed Operations in Compute-in-Memory Architecture". In: *Electronics* 14.7 (2025). ISSN: 2079-9292. DOI: 10.3390/electronics14071340. URL: https://www.mdpi.com/2079-9292/14/7/1340.

[6] NVIDIA Developer. *NVIDIA Isaac*. 2025. URL: https://developer.nvidia.com/isaac.

[7] J. J. Dongarra et al. "An Extended Set of FORTRAN Basic Linear Algebra Subprograms". In: *ACM Transactions on Mathematical Software (TOMS)* 14.1 (1988), pp. 1–17. DOI: 10.1145/42288.42291.

[8] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *CoRR* abs/2010.11929 (2020). arXiv: 2010.11929. URL: https://arxiv.org/abs/2010.11929.

[9] F. El Guibaly. "Design and analysis of arbitration protocols". In: *IEEE Transactions on Computers* 38.2 (1989), pp. 161–171. DOI: 10.1109/12.16493.

[10] John Ellson et al. "Graphviz and Dynagraph - Static and Dynamic Graph Drawing Tools". In: *Graph Drawing Software* (2004), pp. 127–148. DOI: 10.1007/978-3-642-18638-7_6. URL: https://link.springer.com/chapter/10.1007/978-3-642-18638-7_6.

[11] Ella Gale. *TiO2-based memristors and ReRAM: materials, mechanisms and models (a review)*. Sept. 2014. DOI: 10.1088/0268-1242/29/10/104004. URL: https://dx.doi.org/10.1088/0268-1242/29/10/104004.

[12] Simcha Gochman et al. "Introduction to Intel® Core™Duo Processor Architecture". In: *Intel Technology Journal* 10.2 (2006). Intel® Technology Journal, pp. 89–97.

[13] Noam Gozlan, Tom Levi, Marc Louboutin, et al. "Quantum advantage in learning from experiments". In: *Nature* 606 (June 2022), pp. 468–473. DOI: 10.1038/s41586-022-04992-8. URL: https://www.nature.com/articles/s41586-022-04992-8.

[14] Robert M. Gray. "Toeplitz and Circulant Matrices: A Review". In: *Foundations and Trends in Communications and Information Theory* 2.3 (2006), pp. 155–239. DOI: 10.1561/0100000006. URL: https://ee.stanford.edu/~gray/toeplitz.pdf.

[15] Yan-Cheng Guo et al. "CIMR-V: An End-to-End SRAM-based CIM Accelerator with RISC-V for AI Edge Device". In: *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2024, pp. 1–5. DOI: 10.1109/ISCAS58744.2024.10558177.

[16] OpenGenus IQ. *Understanding im2col: A Key Concept in Convolutional Neural Networks*. https://iq.opengenus.org/im2col/. Accessed: 2024-06-15. 2023.

[17] Mike Johnson. *Superscalar Microprocessor Design*. Englewood Cliffs, NJ: Prentice Hall, 1990. ISBN: 9780138756348.

[18] Prabodh Katti, Bashir M. Al-Hashimi, and Bipin Rajendran. *Sparsity-Aware Optimization of In-Memory Bayesian Binary Neural Network Accelerators*. 2024. arXiv: 2411.07842 [cs.ET]. URL: https://arxiv.org/abs/2411.07842.

[19] Riduan Khaddam-Aljameh et al. “HERMES-Core—A 1.59-TOPS/mm2 PCM on 14-nm CMOS In-Memory Compute Core Using 300-ps/LSB Linearized CCO-Based ADCs”. In: *IEEE Journal of Solid-State Circuits* 57.4 (2022), pp. 1027–1038. DOI: 10.1109/JSSC.2022.3140414.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet classification with deep convolutional neural networks”. In: vol. 60. 6. New York, NY, USA: Association for Computing Machinery, May 2017, pp. 84–90. DOI: 10.1145/3065386. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3065386.

[21] Souvik Kundu et al. *CiMNet: Towards Joint Optimization for DNN Architecture and Configuration for Compute-In-Memory Hardware*. 2024. arXiv: 2402.11780 [cs.AR]. URL: https://arxiv.org/abs/2402.11780.

[22] Ann Franchesca Laguna et al. “Hardware-Software Co-Design of an In-Memory Transformer Network Accelerator”. In: *Frontiers in Electronics* 3 (2022). ISSN: 2673-5857. DOI: 10.3389/felec.2022.847069. URL: https://www.frontiersin.org/journals/electronics/articles/10.3389/felec.2022.847069.

[23] Dongjae Lee et al. “Analysis of Data Transfer Bottlenecks in Commercial PIM Systems: A Study With UPMEM-PIM”. In: *IEEE computer architecture letters* 2 (2024), p. 23.

[24] Bing Li et al. “Attar: RRAM-based in-memory attention accelerator with software-hardware co-optimization”. In: *Science China Information Sciences* 68.3 (2025), p. 132401. DOI: 10.1007/s11432-024-4247-4. URL: https://doi.org/10.1007/s11432-024-4247-4.

[25] Can Li et al. “Long short-term memory networks in memristor crossbars”. In: *CoRR* abs/1805.11801 (2018). arXiv: 1805.11801. URL: http://arxiv.org/abs/1805.11801.

[26] Yandong Luo et al. “A FeFET-Based ADC Offset Robust Compute-In-Memory Architecture for Streaming Keyword Spotting (KWS)”. In: *IEEE Transactions on Emerging Topics in Computing* 12.1 (2024), pp. 23–34. DOI: 10.1109/TETC.2023.3345346.

[27] S. Mahadevan et al. “Network traffic generator model for fast network-on-chip simulation”. In: *Design, Automation and Test in Europe*. 2005, 780–785 Vol. 2. DOI: 10.1109/DATE.2005.22.

[28] Linyan Mei et al. “A Uniform Latency Model for DNN Accelerators with Diverse Architectures and Dataflows”. In: *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2022, pp. 220–225. DOI: 10.23919/DATE54114.2022.9774728.

[29] Onur Mutlu et al. “Processing Data Where It Makes Sense: Enabling In-Memory Computation”. In: *ACM Computing Surveys* 52.3 (2019). Covers modern approaches (e.g., PIM, CXL) to mitigate the memory wall., pp. 1–39. DOI: 10.1145/3329785. URL: https://arxiv.org/abs/1903.03988.

[30] Mythic. *M1076 Analog Matrix Processor*. 2025. URL: https://mythic.ai/products/m1076-analog-matrix-processor/.

[31] James W. Nilsson and Susan A. Riedel. *Electric Circuits*. 10th. Pearson, 2015. Chap. 24 (Kirchhoff’s Laws), pp. 35–42, 78–92. ISBN: 978-0133760033.

[32] James W. Nilsson and Susan A. Riedel. *Electric Circuits*. 10th. Pearson, 2015. Chap. 4, pp. 35–42, 78–92. ISBN: 978-0133760033.

[33] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/-Software Interface*. 5th. Morgan Kaufmann, 2017. Chap. 1.4: The von Neumann Model, pp. 23–28. ISBN: 978-0124077263.

[34] Rebecca Pelke et al. *CLSA-CIM: A Cross-Layer Scheduling Approach for Computing-in-Memory Architectures*. 2024. arXiv: 2401.07671 [cs.AR]. URL: https://arxiv.org/abs/2401.07671.

[35] Sourjya Roy et al. *TxSim:Modeling Training of Deep Neural Networks on Resistive Crossbar Systems*. 2021. arXiv: 2002.11151 [cs.LG]. URL: https://arxiv.org/abs/2002.11151.

[36] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0.

[37] Siddharth Samsi et al. *From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference*. 2023. arXiv: 2310.03003 [cs.CL]. URL: https://arxiv.org/abs/2310.03003.

[38] Samsung Semiconductor. *HBM-PIM: Cutting-edge memory technology to accelerate next-generation AI*. 2023. URL: https://semiconductor.samsung.com/news-events/tech-blog/hbm-pim-cutting-edge-memory-technology-to-accelerate-next-generation-ai/.

[39] Ghazi Sarwat Syed, Manuel Le Gallo, and Abu Sebastian. “Phase-Change Memory for In-Memory Computing”. In: *Chemical Reviews* 125.11 (2025), pp. 5163–5194. DOI: 10.1021/acs.chemrev.4c00670. URL: https://doi.org/10.1021/acs.chemrev.4c00670.

[40] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Tech. rep. UCB/EECS-2014-54. Original RISC-V ISA specification (v2.0). University of California, Berkeley, 2014. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.

[41] William A. Wulf and Sally A. McKee. “Hitting the Memory Wall: Implications of the Obvious”. In: *ACM SIGARCH Computer Architecture News* 23.1 (1995), pp. 20–24. DOI: 10.1145/216585.216588.

[42] Furqan Zahoor, Tun Zainal Azni Zulkifli, and Farooq Ahmad Khanday. “Resistive Random Access Memory (RRAM): an Overview of Materials, Switching Mechanism, Performance, Multilevel Cell (mlc) Storage, Modeling, and Applications”. In: *Nanoscale Research Letters* 15.1 (2020), p. 90. DOI: 10.1186/s11671-020-03299-9. URL: https://doi.org/10.1186/s11671-020-03299-9.