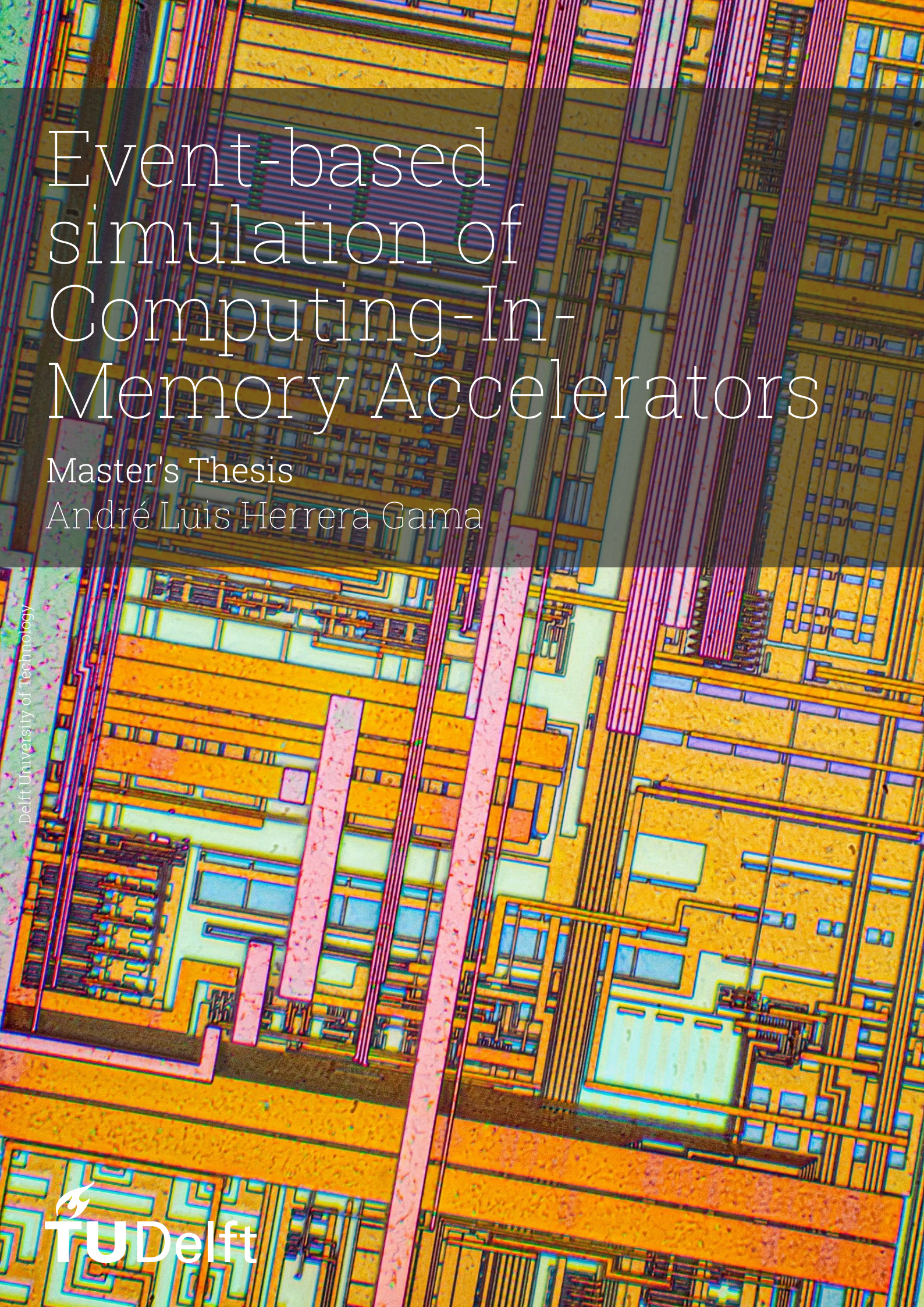


Event-based simulation of Computing-In-Memory Accelerators

Master's Thesis

André Luis Herrera Gama

Delft University of Technology

TU Delft

Event-based simulation of Computing-In-Memory Accelerators

by

André Luis Herrera Gama

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday August 28, 2025 at 11:00 AM.

Student number:	4903439		
Project duration:	January 20, 2025 – August 28, 2025		
Thesis committee:	Prof. dr. ir. G. Gaydadjiev,	Computer Engineering	Chair
	Dr. ir. C. G. M. Verhoeven,	Electronic Circuits & Architectures	Core member
	Dr. T. Spyrou,	Computer Engineering	Advisor

This thesis is confidential and cannot be made public until August 28, 2026.

TU Delft

Preface

This thesis project was a challenging yet rewarding journey. And would not have been possible without the guidance, support, and encouragement of many people in and outside the department, to whom I am sincerely grateful.

First and foremost, I would like to thank Prof. Dr Ir. Georgi Gaydadjiev, my advisor, for his guidance, insightful feedback, and continuous support throughout the development of this work. His expertise and encouragement have not only shaped this thesis but also my professional and personal growth. I would also like to thank Dr. Theofilos Spyrou, my supervisor, for their constructive comments, helpful discussions, and for challenging me to improve the quality of this research. Special thanks go to the PhD students at the Computer Engineering department, especially Konstantinos Stavrakakis and Geerten Verweij, for their insightful discussions, collaboration, stimulating conversations, and supportive atmosphere, which made this journey more enjoyable.

On a personal note, I am deeply grateful to my family for their endless love, patience, and encouragement. I am equally thankful for my partner's constant support and her family's kindness and warmth, all of whom have believed in me and given me the strength to persevere during challenging moments.

André Luis Herrera Gama
Delft, September 2025

Abstract

The hardware demand to accelerate neural network inference in the context of machine learning (ML) and artificial intelligence (AI) has been rapidly growing. Complex models and the widespread use of AI in various domains have led to the problem of energy budget for AI in datacenters becoming a critical problem. The primary contributor is the frequent data movement between memory and processing units, requiring new computing paradigms supporting fast yet energy-efficient neural network inference. Computing-in-Memory (CIM) addresses the "memory wall" of von Neumann architectures by performing parallel computations directly in the memory array. Various memory technologies, including emerging non-volatile memories (NVMs) such as RRAM, FeRAM, PCRAM, STT-MRAM, and optimised structures of conventional SRAM, have been explored for CIM applications.

Designing CIM systems requires multi-level simulation approaches because device or circuit-level design choices can significantly impact system-level efficiency and accuracy. Simulations for CIM architectures span a broad spectrum, from high-level analytical models that provide quick but coarse estimates to detailed circuit-level simulations that provide accuracy at the cost of scalability. However, this leaves a gap in evaluating CIM architectures at realistic workload scales while maintaining sufficient fidelity. This work targets the intermediate cycle- and system-level domain to address this, aiming to bridge the gap between abstract analytical evaluations and low-level hardware description implementations.

This work develops an event-based CIM architecture simulation framework showcased by the simulation of an accelerator system defined with a target multiply and accumulate (MAC) block architecture. Workloads, including basic MLP, CNN, and NLP models, were executed to analyse metrics such as cycle delay, tile count and utilisation as an efficiency indicator. The impact of tunable simulation parameters was also evaluated considering four defined MAC block topologies. Simulation results show for an MLP workload an increase of 7.6% tile utilisation between two topologies, while reporting on both the number of execution cycles and necessary hardware.

Overall, the CIM architecture simulation platform can effectively serve as a tool for mid-stage design exploration and performance evaluation of CIM-based accelerators. The framework's modular, event-based structure enables architectural exploration while providing realistic timing behaviour, distinguishing it from analytical and device-level tools.

Contents

Preface i

Abstract ii

Nomenclature v

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Contribution . . . 2
- 1.3 Outline . . . 3

2 Background 4
- 2.1 Traditional Computing . . . 4
  - 2.1.1 Von Neumann Architecture and Technology Scaling . . . 4
  - 2.1.2 Bottlenecks . . . 4
- 2.2 Hardware Accelerators for data-intensive workloads . . . 5
- 2.3 History of Neural Networks . . . 5
- 2.4 Foundations of CIM . . . 8
  - 2.4.1 Digital memory . . . 8
  - 2.4.2 Analog memory . . . 9
- 2.5 State of the Art Architectures . . . 13
  - 2.5.1 ISAAC . . . 13
  - 2.5.2 PRIME . . . 14
  - 2.5.3 PUMA . . . 15
  - 2.5.4 DREAM-CIM . . . 16
  - 2.5.5 C3CIM . . . 18
- 2.6 Simulation Frameworks Used for CIM . . . 18
  - 2.6.1 GVSoC SoC simulator . . . 21
  - 2.6.2 CIM-Explorer . . . 21
  - 2.6.3 CIM architecture simulation platform . . . 22

3 CIM Simulation Methodology 24
- 3.1 Simulator definition . . . 25
- 3.2 Simulated system organization . . . 26
  - 3.2.1 General system architecture . . . 26
  - 3.2.2 Streaming . . . 28
  - 3.2.3 Quantisation . . . 28
- 3.3 Target architecture implementation . . . 29
  - 3.3.1 Signed arithmetic . . . 30
  - 3.3.2 Simulated metrics . . . 31
  - 3.3.3 Pipeline . . . 31
  - 3.3.4 Tiling organization . . . 31
  - 3.3.5 Utilisation . . . 32
  - 3.3.6 Bounded problem . . . 33
  - 3.3.7 Unbounded problem . . . 33
  - 3.3.8 Generalisation to other architectures . . . 33
- 3.4 Tunable parameters . . . 33
- 3.5 Simulated workloads . . . 35
  - 3.5.1 Mapping CNN into GEMM . . . 35
  - 3.5.2 Choice of ANN models . . . 36

4 Simulation results **39**
4.1 Functional architectural verification . . . 39
4.1.1 Simulating delays . . . 40
4.2 Workload Evaluation . . . 44
4.2.1 Simple MLP workload . . . 44
4.2.2 LeNet . . . 45
4.2.3 MobileNet . . . 45
4.2.4 DistilBERT . . . 46
4.2.5 Comparison . . . 46
4.3 Simulator Performance . . . 49

5 Conclusion **50**
5.1 Future prospects . . . 51

References **52**

Nomenclature

Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
DNN	Deep Neural Network
ML	Machine Learning
MLP	Multi-Layer Perceptron
NLP	Natural Language Processing
SNN	Spiking Neural Network
CIM	Computing in Memory
GPU	Graphical Processing Unit
HDL	Hardware Description Language
DRAM	Dynamic Random Access Memory
FeRAM	Ferroelectric Random Access Memory
NVM	Non-Volatile Memory
PCRAM	Phase Change Random Access Memory
RRAM	Resistive Random Access Memory
SRAM	Static Random Access Memory
STT-MRAM	Spin-Transfer Torque Magnetic Random Access Memory

1 Introduction

This chapter introduces the topic of this thesis. First, the chapter discusses the topic of computing in memory (CIM) and its simulation, and then presents the state of the art and the main thesis contributions. At the end, an outline for the remaining chapters of this document is presented.

1.1. Motivation

The fast development of new computing paradigms driven not only by the growing popularity of artificial intelligence (AI) and Machine Learning (ML) [1] but also by increasing demands for high-performance, energy-efficient and scalable computing [2], has fueled the search for alternative memory technologies. CIM has emerged as a promising approach to address the "memory wall" challenge [3]. In traditional von Neumann architectures, computation and data storage separation lead to latency and energy overheads for data-intensive tasks because large volumes of data must be frequently transferred between memory and processing units. CIM enables in-situ parallel processing by allowing the memory to perform simple computations, such as logic AND operations, significantly improving throughput and reducing energy consumption. At the core of AI/ML computation, multiple matrix-vector multiplication (MVM) operations consider stationary synaptic weights and an input to be processed [4]. CIM allows synaptic weights to be written once to modified memory cells and used in numerous parallel multiply-and-accumulate (MAC) operations.

The choice of memory technology is the first step to implementing CIM effectively. Emerging non-volatile memories (NVMs) such as Resistive RAMs (RRAMs) [5] offer high-density and CIM capabilities through resistive switching, enabling efficient parallel MAC operations. At the same time, digital memories, such as modified SRAM arrays [6], offer mature CMOS compatibility and predictable performance for near-future solutions [7].

During the creation of architectures to be deployed in real-world systems, extensive design-space exploration (DSE) and performance evaluation are essential [8], where simulation plays a critical role [9].

Given the numerous trade-offs and design choices in each of these designs and the constant development of analog memory technologies [6], system modularity when designing such simulators is of great importance. Different memory technologies that could be simulated and compared using the same accelerator system include Ferroelectric RAM (FeRAM) [10], which offers low write energy but limited scalability; Spin-Transfer Torque Magnetic RAM (STT-MRAM) [11], known for high endurance but relatively higher write latency; and Phase-Change RAM (PCRAM) [12], which provides multi-level storage capability but suffers from high energy consumption for write operations.

Therefore, a simulation platform capable of streamlining DSE while incorporating realistic physical constraints is essential for the future of CIM accelerators, especially as these systems become prototypes rather than theoretical assumptions and must meet market and power demands for efficient computing.

Various CIM designs and systems have been proposed for emerging memory analog crossbar arrays [13, 14, 15] and fully digital CIM accelerators [6, 16]. Analog CIM solutions, particularly based

on RRAM [17], offer high parallelism and density but deal with significant challenges regarding variability, endurance, and integration complexity [5]. Additionally, they require power-hungry and area-intensive data converters, such as analog-to-digital (ADC) and digital-to-analog converters (DAC) [14] to interface with the rest of the digital computing system, which can reduce their efficiency gains.

Usually built on SRAM memory technology, digital CIM designs offer well-known deterministic operation while leveraging mature CMOS fabrication technologies [6]. However, they are naturally limited to storing a single bit per cell, limiting the numerical precision of in-memory computations compared to analog solutions. Other than that, partial sums in digital CIM are accumulated outside the memory array [18], reducing the parallelism compared to the promise of its analog counterparts and increasing data movement. Furthermore, SRAM has a lower density due to the necessary power connections and the number of transistors that implement its coupled inverters.

These technology trade-offs are important considerations when tailoring CIM applications. For this reason, designers can simulate and analyse different designs to guarantee maximum efficiency, considering the boundaries of CIM technologies, as part of their research.

Simulation frameworks are indispensable in evaluating these architectures before fabrication. On the lower level, existing CIM simulations rely on general circuit simulators such as Cadence Spectre [19], which provide detailed circuit modelling using non-linear models [20] that lead to high computational cost. On the higher level, analytical frameworks enable fast exploration across large design spaces, but inevitably sacrifice timing fidelity and architectural detail. Analytical simulators include ZigZag [21], which focuses on memory hierarchies and mapping of neural networks; CIMLoop [22], which supports compiler-level design of CIM accelerators; NeuroSIM [23], which estimates energy and area across device and architecture levels; and CIM-E [24], which accelerates DSE of RRAM-based CIM systems using device-specific trends.

At the system level, simulators capable of modelling entire processors and SoC platforms, such as Gem5 [25] and GVSoC [9], are widely used. However, they need to be manually extended to incorporate accelerator components. For being cycle-accurate, Mnemosene [26] provides detailed timing models tailored to device effects for CIM-specific simulation, but its focus limits efficiency for broader architectural exploration.

Event-based simulation covers a compelling middle ground [25, 9], because it enables accurate timing modelling, allows parallelism, and supports integration of memory-access patterns while remaining computationally efficient, making it particularly suitable for architectural exploration of CIM-based designs. Event-based simulators also do not require licensed process technology files, as their modular structure relies on user-defined delays and standardised interfaces.

1.2. Contribution

While CIM appeared as a promising paradigm for overcoming the memory wall in various data-intensive workloads, current research often focuses on device-level demonstrations or high-level algorithmic evaluations. However, a simulation framework that combines detailed timing fidelity with architectural flexibility is missing, which would enable systematic DSE across a wide range of CIM-based accelerator systems. Existing tools are often limited in scope, tied to specific memory technologies and designs, or incapable of simulating event-driven behaviour at the system-accelerator level.

This work addresses these gaps by presenting an event-based simulation framework for CIM architectures, which is demonstrated using a digital 8T-SRAM implementation [16] and a simple accelerator system architecture. The framework enables cycle-accurate modelling of CIM operations, supports flexible configuration of architectural parameters, and facilitates performance evaluation under diverse workloads. While the framework can be easily scaled to support a variety of memory types, such as the targeted architecture [16], the simulator is designed to capture the specific timing and operational characteristics of a CIM accelerator system, making it a valuable tool for researchers and designers.

The main contributions of this thesis can then be summarised as:

- Definition of system requirements for a CIM accelerator system, considering the targeted tile microarchitecture and its system definition;

- Implementation of an event-based CIM architecture simulator platform;
- Demonstration of the platform's support for multiple workload types and flexibility through defined tunable parameters.

1.3. Outline

The remainder of this thesis is organised as follows: Chapter 2 provides a background on traditional computing, hardware accelerators, neural networks, computing-in-memory foundations, state-of-the-art architectures, and simulation frameworks. Chapter 3 defines the simulator, details the simulated system organisation, including assumptions used when simulating the target architecture, and discusses the workloads used for testing and comparison. Chapter 4 presents experimental results, verification, a comparison using different workloads and system parameters, and performance evaluation. Chapter 5 summarises the contributions and outlines potential directions for future research in CIM simulation.

2

Background

This chapter presents the theoretical background relevant to CIM and data-intensive workloads. It reviews the traditional computing landscape, highlighting its architectural features and limitations. Next, it introduces hardware accelerators designed for data-intensive applications. Finally, it provides the necessary background on neural networks (NNs), which are the primary driver for the development of CIM architectures.

2.1. Traditional Computing

Over the past decades, the von Neumann architecture has become the foundational model for traditional computing systems. Such architecture defines the separation of memory and processing units. While this design has enabled decades of progress, it also introduces limitations that affect performance in modern, data-intensive applications.

2.1.1. Von Neumann Architecture and Technology Scaling

Since von Neumann's report [27], its architecture type has been the primary driver of computer development throughout history. Such architecture starts its definition with a central processing unit (CPU) that receives inputs and produces outputs. Inside this CPU, a control unit and an arithmetic/logical unit transform the input with the assistance of a memory unit. To enable general-purpose computing, instructions and data are stored in the same memory, simplifying programming.

Apart from that, other technology projections also dictate the current computing landscape. In 1965, Gordon Moore projected an exponential growth rate of two per year in the number of components per integrated circuit, known as Moore's law [28]. Moore initially claimed that the scaling would hold at least until 1975, but the prediction still roughly holds until today. Only in the most recent advancements is it possible to see a deceleration of this growth due to physical and economic limits of transistor sizing.

Together with Moore's law, Dennard scaling [29] has further intensified performance gains in recent decades. Dennard scaling, also called Dennard's Law or MOSFET scaling, states that as metal-oxide-semiconductor field-effect transistor (MOSFET) dimensions diminish, so does power consumption. This law not only holds for a power consumption perspective, but also relates to how transistors could run faster and cost less as they shrink. Since 1974, this has held for larger-sized transistors. However, as transistors shrink even more, so does their dielectric layer, leading to shorter channels. Consequently, leakage currents became increasingly more relevant during design and use, and Dennard scaling effectively ended between 2005 and 2007.

2.1.2. Bottlenecks

Von Neumann architectures prioritise sequential execution and data movement from memory to the processor. However, in modern, data-intensive workloads, this can lead to diminishing performance returns, as the data transfer between memory and compute units becomes a significant bottleneck, a phenomenon called "memory wall". In many cases, moving data consumes more time and energy than

the actual computation.

This scenario is aggravated by the slowing of Moore's Law and the end of Dennard scaling. Although transistor sizes have continued to shrink, power consumption has not decreased proportionally, leading to increased power density and thermal challenges. As a result, not all parts of a chip can be powered simultaneously, a limitation known as "dark silicon" that constrains the effective use of available hardware resources [30]. CIM architectures address this limitation by decentralising computation with its memory arrays, minimising data movement, and reducing reliance on high-power processor cores, enabling more operations within thermal and power budgets imposed by dark silicon.

2.2. Hardware Accelerators for data-intensive workloads

The growing demand for processing large batches of data has extrapolated the limits of traditional computing architectures. As a result, specialised hardware targeting specific types of workload has emerged as a key solution. These accelerators are designed to optimise computation and data movement for their target tasks, providing efficient support for data-intensive applications such as AI.

The recent popularity of AI and ML can be attributed to the appearance and rapid spread of Large Language Models (LLMs) and the use of AI in various fields. The foundation of AI lies in neural networks, which are well-known amongst supervised learning algorithms. At the core of these learning algorithms are matrix-vector multiplications (MVM), simple operations involving parallel multiplications and sequential additions done to many numbers. This type of algorithm has a high level of parallelism and, most importantly, requires fast and large storage access.

Specialised hardware is used to run such parallel and memory-intensive algorithms while keeping in mind performance and power requirements. A prominent example of these is Graphical Processing Units (GPUs). GPUs are optimised for highly parallelisable tasks, leveraging multi-core performance and providing an external and fast storage unit for these data-intensive tasks. As the name suggests, GPUs are known for processing fast and neat graphics for multiple computer applications.

Because of their fast storage capabilities, GPUs have been good candidates for training and inference of AI models. However, there is still a need for specialised hardware that can operate more efficiently, faster or in constrained environments. Such specialised pieces of hardware are called hardware accelerators, designed to perform specific functions, leaving other types of computation to general-purpose CPUs. Examples include Tensor Processing Units (TPUs), tailored for matrix operations in ML, and other domain-specific accelerators targeting applications beyond graphics.

Even specialised hardware suffers from some limitations common to von Neumann architectures. Being "Data movement" the most relevant limitation for this thesis project. Moving large amounts of data from the memory unit to the computing unit and then back to the memory unit requires an excessive amount of time and energy. In inference, for example, the computation itself is elementary but has to be applied repeatedly to various data strings.

2.3. History of Neural Networks

Although neural networks are highly relevant right now due to AI, they are not a new concept. As early as 1943, McCulloch et al. [31] introduced the theoretical foundation of neural computation models. Their mathematical model laid the groundwork for the perceptron, as described by Rosenblatt et al. [32], one of the earliest practical implementations of a neural network, even considering the significant computational limitations in 1958. These early efforts represent the first generation of neural networks, characterised by simple neuron models that activate when an input exceeds a predefined threshold. Because these are not biological neurons, a common name for them is Artificial Neural Networks (ANN), an umbrella term for what is discussed in this section, but most commonly used to refer to second-generation networks. Due to their threshold-based activation, first-generation neural networks are called Threshold Logic Units (TLUs). These networks relied on simpler mechanisms and logic for learning and could only solve linearly separable problems due to their single-layer topology, limiting their real-world applications.

In 1986, Rumelhart et al. [33] introduced Multilayer Perceptrons (MLPs) and the backpropagation algorithm, which marks the beginning of what is known as the second generation of neural networks.

These second-generation networks are commonly referred to using a broader term: Feedforward Neural Networks (FNNs). They are composed of multiple layers of neurons where information flows in one direction without any cyclic behaviour.

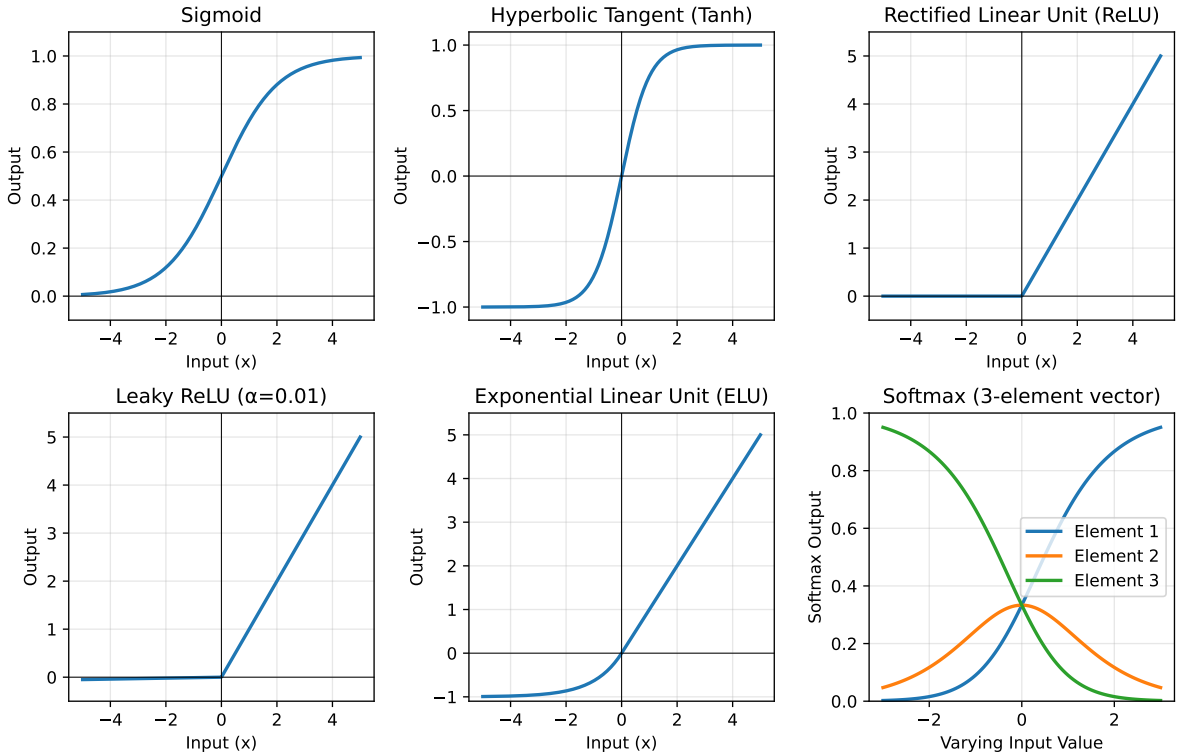


Figure 2.1: Activation functions overview.

Unlike the first generation, which relied on binary threshold units, these networks employed non-linear activation functions such as `sigmoid`, Rectified Linear Unit (`ReLU`) and hyperbolic tangent (`tanh`), enabling networks to approximate complex, non-linear functions. This architectural shift significantly increased workload complexity, as training Deep Neural Networks (DNNs) (such as MLPs and Convolutional Neural Networks (CNNs)) required multiple operations over large datasets, involving forward and backwards propagation.

Figure 2.1 shows a graphical representation of activation functions, followed by some of their derivatives in Figure 2.2. Amongst these activation functions, the `sigmoid` maps inputs to a [0, 1] range, making it useful for probabilistic interpretations but prone to the exponential decay of gradients in training when inputs fall in saturated regions. Similarly to `sigmoid`, the `tanh` compresses values into the [-1, 1] range, which helps centre the data around zero and can lead to faster convergence; however, it still suffers from the same exponential gradient decay. `ReLU` mitigates the gradient decay problem and enables efficient training of deep networks by outputting zero for negative inputs and keeping the value for positive ones; however, its tendency to output zeros for negative inputs can lead to what is called a “dead neuron”, where the neuron naturally stops its contribution from the moment it becomes zero. Variants such as Leaky `ReLU`, which assigns a small slope to negative inputs, and Exponential Linear Unit (ELU), which smooths out negative values toward -1, were introduced to reduce the number of dead neurons and improve gradient flow, but are computationally expensive. Lastly, the `softmax` activation is commonly used in the output layer of classification networks, because it converts raw scores into normalised probability distributions over classes, allowing the network to make user-interpretable predictions.

In 1998, LeCun et al. [34] further amplified computational demands with the advent of CNNs, where spatially-localised operations introduced high data reuse but also increased the need for efficient memory access patterns. The breakthrough of deep learning models, exemplified by Krizhevsky et al.’s AlexNet [35], shifted workloads to a new scale since 2012, necessitating specialised hardware such as GPUs to handle the compute and memory demands. As summarised by LeCun et al. [1] in 2015,

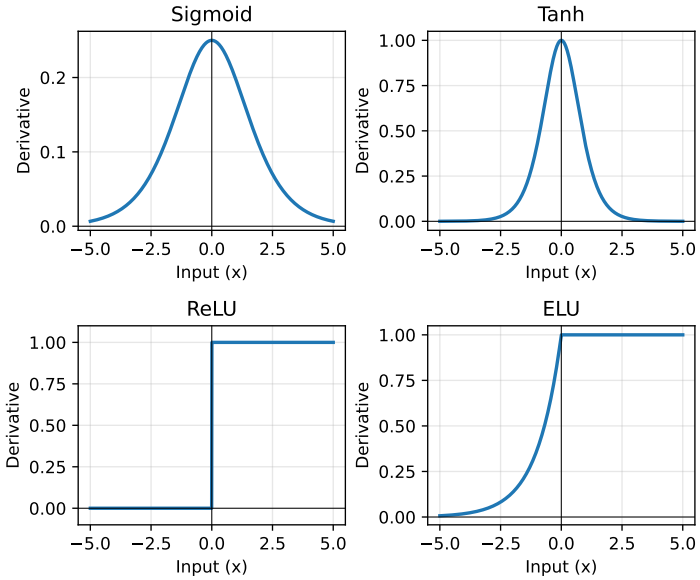


Figure 2.2: Derivative of activation functions.

modern neural networks have reached a point where data movement is the primary bottleneck, not computation, influencing how workloads interact with hardware resources. Recurrent Neural Networks (RNNs) [36], introduced for sequential data modelling, expanded the scope of neural networks to time-series and natural language tasks. More recently, Transformer architectures, popularised by Vaswani et al. [37], have changed sequence processing with attention mechanisms, eliminating the sequential bottleneck of RNNs and enabling LLMs.

As neural network workloads continue to evolve, emerging architectures like Binary Neural Networks (BNNs) [38], Ternary Neural Networks (TNNs), and Spiking Neural Networks (SNNs) [39] present new computational paradigms. The first two models reduce data precision, bringing opportunities for the current landscape of hardware accelerators. However, SNNs introduce event-driven sparsity, which challenges existing accelerators that rely on synchronous execution of some components.

While traditional neural networks process information through continuous activations, SNNs [39] represent a paradigm shift by mimicking the event-driven communication patterns of biological neurons. Unlike the dense and synchronous computations of MLPs and CNNs, SNNs operate on discrete spikes—binary events triggered when a neuron's membrane potential crosses a threshold. These spikes enable inherently sparse and truly asynchronous computation, reducing the number of operations and data movements required, but also reducing predictability. A key element in this architecture is the Leaky Integrate-and-Fire (LIF) neuron model, which accumulates incoming spikes over time, while gradually decaying the neuron membrane potential (leaky integration) until it reaches a firing threshold. The introduction of this time-independent dataflow and trigger-based processing in SNNs has led to their classification as the third generation of neural networks, which offers potential for energy-efficient computing. This energy efficiency comes from the reduced computations that these models might present. However, these sparse and irregular workload patterns bring unique challenges for traditional or modern hardware accelerators optimised for dense matrix operations. There exists then the need for specialised architectures capable of exploiting SNNs' distinct computational characteristics.

As the characteristics for hardware accelerators for SNNs are bound to have very different topologies and working mechanisms, this thesis uses conventional DNNs, which represent the prevailing generation of neural network models. Since SNNs are still new and immature in computational workloads, their implementation, simulation and mapping to CIM paradigms could also lead to a stand-alone thesis project. An overview of the most well-known neural network terms can be seen in Figure 2.3. The figure shows how the different neural networks described can be grouped together and separated into the three different generations, highlighting how initially, second-generation neural network models are also implemented as SNNs.

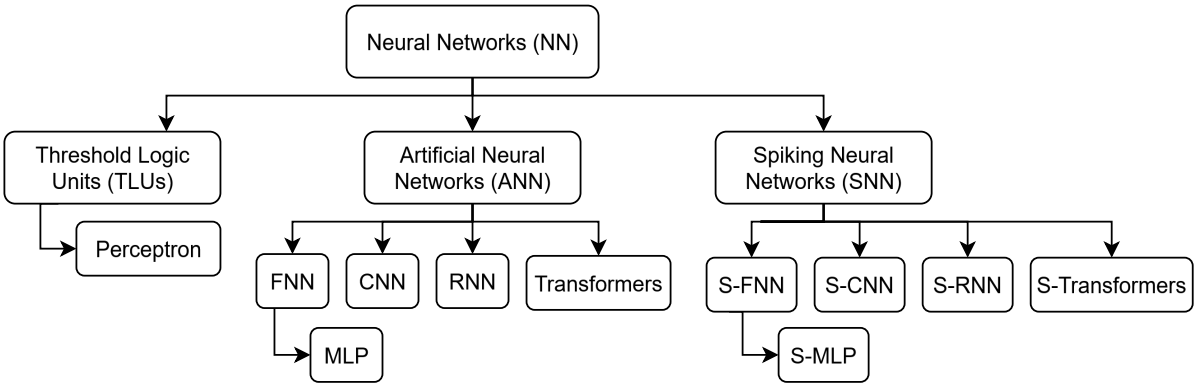


Figure 2.3: Neural networks overview where spiking model variants are indicated with “S-”.

2.4. Foundations of CIM

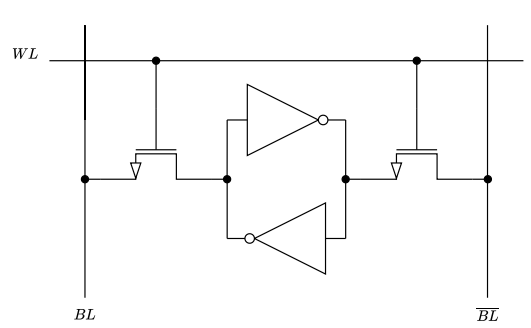
This section discusses the types of CIM architectures, including digital and analog memory technologies. Then, it provides an overview of CIM architectures introduced over the years.

CIM is a growing non-von Neumann computing concept that aims to fill the gap between the performance and demand of neural network inference. Tackling specifically matrix multiplication, CIM cores are designed to use significantly less energy for comparable neural network workloads. This efficiency is not only crucial for high-performance systems but also for small-scale embedded systems [40].

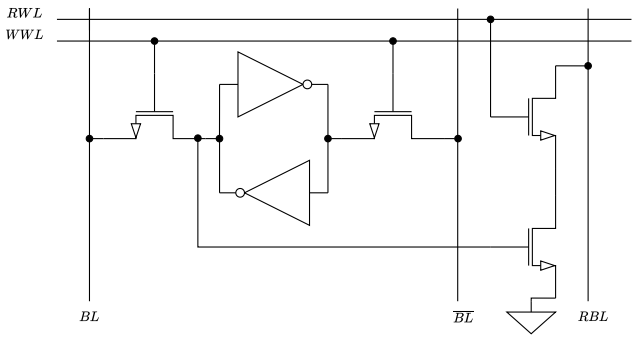
Different types of CIM architectures were created following advancements in memory technology over the years. On a top level, computing memories can be classified into digital and analog approaches. Each offering its advantages in terms of scalability, writing and reading times and energy efficiency.

2.4.1. Digital memory

Derived from existing technologies such as Static Random Access Memory (SRAM). Digital memories allow computing with a modified memory cell. Instead of the standard 6-transistor (6T) memory cell architecture seen in Figure 2.4a, more transistors and wires allow for bitwise AND operations in the memory cell. An example of such a design is shown in Figure 2.4b.



(a) 6T-SRAM, based on [41].



(b) 8T-SRAM, Based on [16].

Figure 2.4: Full-CMOS SRAM memory cells.

As used for the DREAM-CIM MAC block design [16], the small addition of two transistors results in a “free” AND (bitwise multiplication) operation. The cost of operating the memory cell is the addition of two extra transistors, one extra bitline (BL), named Read Bitline (RBL), and one extra wordline (WL), named Read Wordline (RWL). For the AND operation to occur, the first bit of the input operand is provided through the RWL, while the second is the one that was previously stored in the memory cell. The layout also allows for parallel reads, where the RBL carries the result of the operation on a read. The accumulation is then done with different adder configurations depending on the CIM design.

Unlike analog memories, digital memories neither allow for the accumulation of partial products nor a

higher bit multiplication per cell. However, it provides the following benefits over analog memories:

- Does not require DACs and ADCs that increase area and design complexity;
- Have the flexibility of shorter writing times, which allow for more frequent changes in memory content, and consequently, different workload mapping and DSE parameters;
- Provide much better noise tolerance, including robustness to thermal noise variations [42].

2.4.2. Analog memory

Nowadays, standard memory technologies such as SRAM, DRAM and Flash rely on storing data (bits) in terms of charge. SRAM stores charge in cross-coupled inverters, DRAM in a capacitor and Flash in a metal-oxide semiconductor (MOS) floating gate. Scaling limitations regarding charge storage are expected in the future, either by physical limits in transistor sizing for SRAM and Flash or unrealistically deep "deep trench" capacitors for DRAM [43]. Amongst the current commercialised memory technologies, Flash is the only one that does not require constant power delivery or frequent refresh. However, this changes when new analog memory technologies are introduced:

- Ferroelectric Random-Access Memory (FeRAM): Utilises the polarisation of the ferroelectric film, which is placed between two electrodes to store data. If the film is upwards polarised, the FeRAM device stores a 0; otherwise, a 1 [10]. FeRAM is the only memory with destructive read from this list and offers low write energy at the cost of limited scalability;
- Spin-Transfer Torque Magnetic Random-Access Memory (STT-MRAM): Similar to FeRAM, STT-MRAM relies on polarisation, but instead of electric, the polarisation is magnetic. The device comprises two ferromagnetic layers, one free and one reference, separated by a tunnel barrier. During a read, the magnetisation of the layers affects the device's resistance; if the polarisation is parallel (same direction), the resistance is low, and otherwise, high [44]. STT-MRAM offers high endurance but higher writing latency;
- Phase-Change Random-Access Memory (PCRAM): As the name suggests, the phase-change of a chalcogenide material happens between the crystalline and amorphous states via a temperature change. Which, in turn, leads to low and high resistance states respectively [45]. PCRAM was observed to support different storage levels, but the technology suffers from high write power consumption due to the current needed to raise the temperatures;
- Resistive Random Access Memory (RRAM): RRAMs are based on a metal-insulator-metal structure commonly named memristive devices. It is one of the most studied emerging memory technologies; therefore, its functionality is discussed in detail in the following subsection.

Table 2.1: Performance comparison of emerging non-volatile memories (NVMs).

Memory Type	Read Time	Write Time	Endurance
FeRAM [10]	50–100 ns	50–100 ns	$\sim 10^{12}$ cycles
STT-MRAM [44]	$\sim$10 ns	10–50 ns	$> 10^{15}$ cycles
PCRAM [45]	10–50 ns	50–500 ns	10^{6}–10^{8} cycles
RRAM [5]	$\sim$10 ns	10–100 ns	10^{6}–10^{9} cycles

Values exemplifying the comparing metrics for the technologies mentioned above are shown in Table 2.1. It is possible to see how RRAM and STT-MRAM present a good reading time amongst the options, making them two relevant technologies to be explored for CIM. Furthermore, it is possible to see how writing times are around one order of magnitude longer than reading times, making these memories not suitable for many rewriting operations.

Memristive devices

Memristive devices are the building blocks for RRAMs; the memristive concept was first described half a century ago, in 1971 [47]. In this work, Chua introduced the ideal memristor as the "missing component", defined by a relation between magnetic flux and charge, thereby completing the set of four fundamental two-terminal circuit elements. This relationship is illustrated in Figure 2.5.

To account for physical devices that do not strictly obey this ideal relation, Chua later generalised the concept to the broader class of memristive systems in 1976 [48]. HP Labs demonstrated a memristive

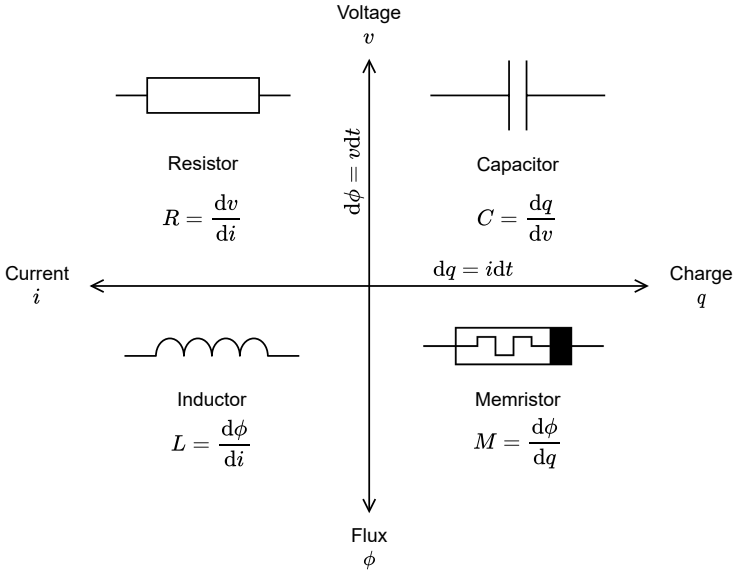


Figure 2.5: Four fundamental two-terminal circuit elements, adapted from [46].

device only in 2008, using a TiO_2-based resistive switching structure [46]. Although widely referred to as the “HP memristor”, this device is more accurately understood as a memristive system rather than an ideal memristor. The HP memristive device is the building block used in today’s RRAM technology.

Memristive devices are composed of a top (TE), a bottom metal electrode (BE), and a doped insulator material in between, most commonly, a metal oxide. Through its lifetime, memristive devices go through six stages, two during fabrication and four during normal operation:

- Recently fabricated: The state where freshly deposited metal oxide contains a few oxygen vacancies and needs more to be created by the next stage;
- Forming: A high voltage difference causes the negative oxygen ions (anions) to drift towards the boundary with the TE, leaving oxygen vacancies that form a Conductive Filament (CF) between TE and BE;
- Low resistive state (LRS): Occurs when the memristive device has the most oxygen vacancies and high conductivity. It is considered a logic 1 for single-bit cells;
- Reset: Sufficiently low voltage is applied to the memristive device, where oxygen anions start to return to the vacancies;
- High resistive state (HRS): After anions fill some vacancies, the memristive device is less conductive and therefore in HRS. HRS is considered a logic 0 for single-bit cells;
- Set: When a sufficiently high voltage (but lower than forming) is applied for a controlled period, more vacancies appear, until the memristive device is again in the LRS.

The analog behaviour of the memristive device can be seen in the I-V characteristic curve in Figure 2.6, where it is also possible to deduce, with the help of the arrows, the four modes of regular operation: Set, HRS, Reset, and LRS.

The current way to simulate and include memristive devices in the DSE of CIM architectures is by using a model. These models mathematically mimic the memristive device response to an applied voltage. Different models exist depending on the level of specificity in simulation. A commonly used model for circuit-level simulation is the Verilog-A JART VCM v1b [20], developed by the electronic materials research laboratory, which is a collaboration between RWTH Aachen and Forschungszentrum Jülich.

Memory cell topology

Memristive devices are connected into arrays to form a memory block. The density of the memory array depends on the type of cell topology, which can be:

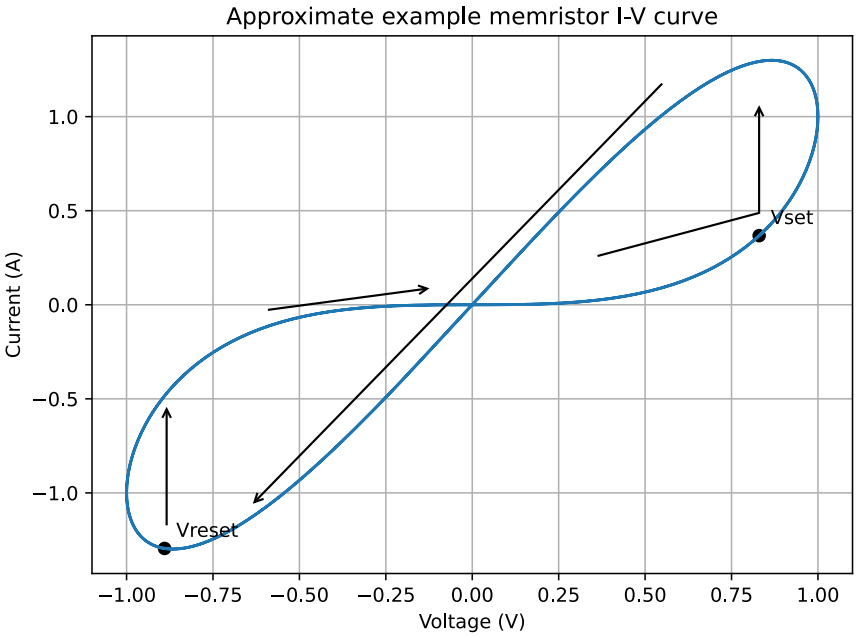


Figure 2.6: Example of Memristive device I-V curve, adapted from [46].

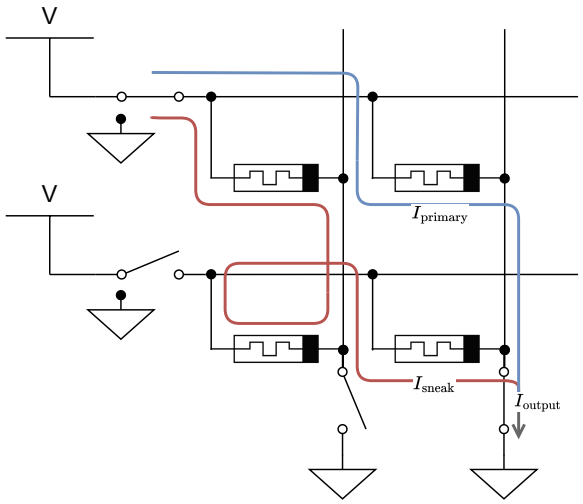


Figure 2.7: Sneak-path example on a 2x2 1R RRAM crossbar array, adapted from [49].

- 1R: Memristive devices are placed between bitlines and wordlines, forming what is popularly called a crossbar. Where memristive devices connect access lines perpendicularly, this configuration allows for a very high memory density, making great use of the small scale of memristive devices. However, accessing the cells, especially in parallel, becomes challenging because of a phenomenon called “sneak paths”. Sneak paths are voltage loops created through other memristive devices that are not meant to be written to or read from, causing a disturbance in the total expected current through the line. This disturbance challenges memory design and sets a boundary on the maximum size of crossbar arrays that can be reliably read and written. An example demonstrating sneak paths for a minimal array is shown in Figure 2.7, where the red line shows an alternative current path passing through three memristive devices that is then added to the current from the target device. From the figure, it is also possible to deduce how more RRAM cells would allow for more sneak-paths to contribute to the total output current;
- 1D1R: A diode in the memory cell prevents sneak currents from influencing the final output current.

It can be considered a smaller addition compared to a transistor. However, the drawback of this design is the limitation on using only unipolar switching memristive devices [50];

- 1T1R: Similar to a DRAM cell, this topology has one access transistor that allows for smoother operation, preventing sneak-paths and adding voltage control at the cost of decreasing the density of the memory array. This cell topology requires a new line per column, commonly known as the Source Line (SL);
- 2T1R: Used by Biyani et al. [51], this relatively new cell architecture has two transistors to control the reads and writes to the memristive device. This topology was observed to increase the overall power efficiency of the RRAM array, at the cost of slightly higher cell area. This cell architecture's in-depth functionality is presented with a CIM design in section 2.5.

Analog dot product and sum

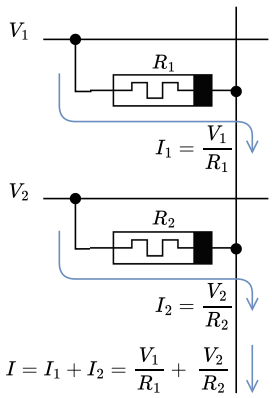


Figure 2.8: Bitline sum of cell products, adapted from [13].

Unlike digital CIM, which only allows for a 1-bit dot product (AND) operation by varying the read voltage, analog memories also allow for accumulation in the crossbar. This operation can be seen in Figure 2.8. The Multiply and Accumulate (MAC) operations are based on Ohm's and Kirchhoff's Law, where V_1 and V_2 represent one vector, and R_1 and R_2 represent the other; they are multiplied and their products are accumulated in the bitline by summing up the two resulting currents. DACs are needed in the wordlines to provide voltage levels that translate the input vector. Similarly, ADCs are used on the bitlines to interpret the MAC operation output. This operation is the baseline for resistive memory technology, allowing for highly parallel MVM.

Bit precision

Another active field of research regarding analog memory is the bit precision of the input and weights stored in the crossbar. Theoretically, multi-level DACs allow multi-bit inputs, while different cell resistances allow multi-bit weights, where the transformation back to digital by the ADC must also be supported at multiple levels.

Variability

in memristive devices [52, 53] plays a big role in using resistive memory cells for CIM architectures. Variability can be categorised as:

- Cycle-to-cycle (C2C): Refers to the inherent randomness in filament formation and rupture [54], during SET and RESET operations, which also leads to different perceived resistances in the HRS and LRS states;
- Device-to-device (D2D): As the name suggests, it comes from manufacturing variability, such as differences in material area, thickness and concentration [55]. D2D variability leads to a difference in the spatial variation of resistance and required SET and RESET voltages. The reading/writing circuit can be changed to mitigate such changes, but requires a great area overhead and design complexity.

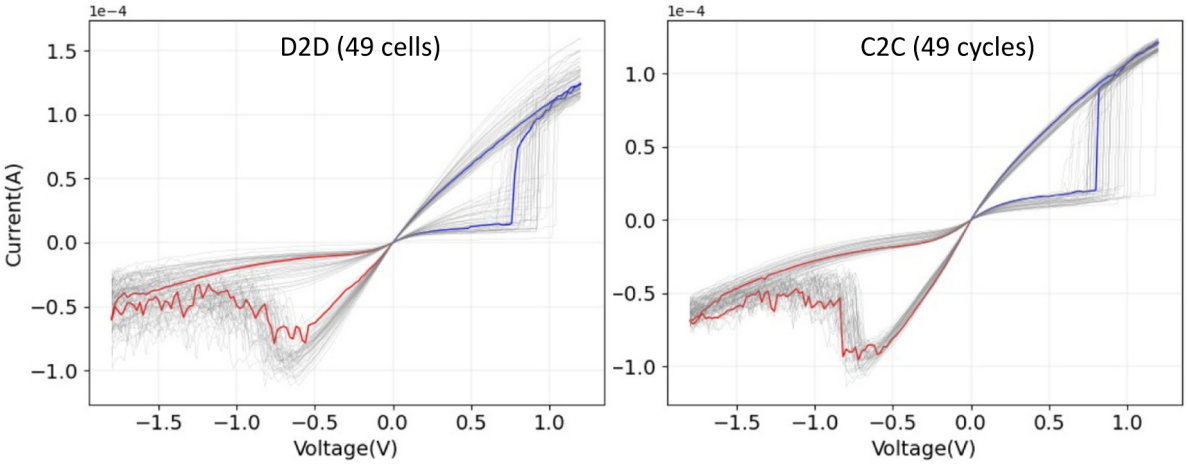


Figure 2.9: Experimental cell-level D2D and C2C variability with highlighted nominal characteristics [52].

Figure 2.9 shows experimentally how memristive device variability affects the I-V curve of a nominal operation range.

Variations in V_{HRS}/I_{LRS} have even been used as seed for random number generation [56], due to the true random variation in different RRAM devices and their cycles. However, minimal variation is desired for memristive devices in CIM, even if random. CIM architecture design is highly affected by device variability. Since analog CIM leverages the parallelism in crossbars, compound changes can lead to wrong reads, an unacceptable fault in electronics design. This variability leads to various current CIM architectures to use a single-bit representation per RRAM cell, maintaining design reliability.

2.5. State of the Art Architectures

This section presents the state of the art in analog and digital CIM architecture designs. As discussed in section 2.2, new architectures are being constantly developed to supply the growing demand for energy-efficient neural network engines.

Given the numerous trade-offs and design choices in each of these designs and the constant development of analog memory technology, system modularity when designing such systems is of great importance. A simulating platform capable of streamlining DSE without leaving behind physical limitations is what will set the future of CIM accelerators. Especially as these systems scale and have to meet market and power consumption demands for the future of computing.

2.5.1. ISAAC

In 2016, Shafiee et al. [13] proposed ISAAC (In-Situ Analog Arithmetic in Crossbars) for the inference process of CNNs. The work aims to extend previous computing near-memory work from DaDianNao [57], to which most metrics are compared. Because the paper proposes a full-fledged accelerator design based on crossbars, and it is one of the first to do so, it gained rapid popularity and is, until the end of this thesis, still used as a benchmark for newer CIM accelerators.

Shafiee et al. [13] choose the 1T1R cell topology for their design due to its functionality in avoiding sneak paths. Due to its stability, this design allows for a higher bit-density per cell, which is chosen to be two, while keeping the input with 1-bit density for a simple DAC design (an inverter) and leading to a 2-bit ADC.

Figure 2.10 illustrates the architectural hierarchy, which is organised into four levels: chips at the top, followed by “Tiles”, IMAs (MAC cores), and finally crossbar arrays.

Analog memory is known for long writing times compared to other memory technologies. Therefore, the design follows a “minimal writing” scheme, partitioning CNN layers amongst different “Tiles”, avoiding rewriting to conserve time and decrease energy consumption. Another main paper contribution is the pipeline design, which reduces buffer sizes and brings credibility to the architecture.

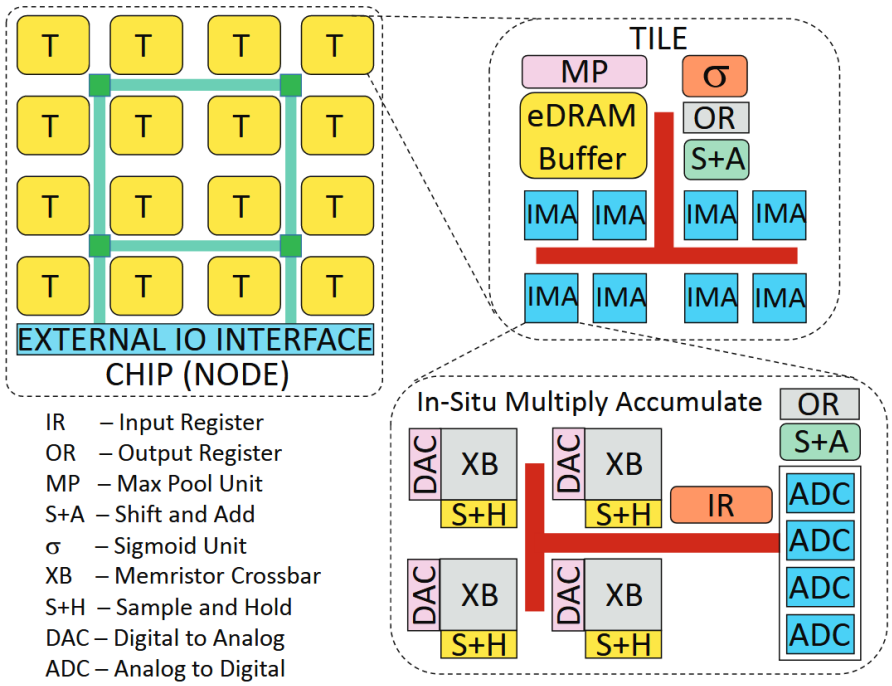


Figure 2.10: ISAAC architecture hierarchy [13].

Overall, the paper further describes their found optimal parameters for their architecture with eight crossbars of size 128×128 per "Tile", and how the workload is distributed for optimal pipelining. The algorithmic pipeline balancing and communication structure are out of this thesis project's scope. However, the added support for pipeline simulation is essential for CIM simulation and falls under the scope.

ISAAC uses energy and area models from:

- CACTI 6.5 [58]: For buffers and interconnects;
- Dot-product engine [59]: Crossbar;
- Adapted from DaDianNao [57]: Shift-and-add circuits, max-pool circuit, sigmoid operation and off-chip links (HyperTransport serial link model).

Furthermore, despite multiple different sources for their models, no centralised simulator was mentioned to be used in the design process.

2.5.2. PRIME

Proposed by Chi et al. [15], PRIME is proposed with a software-hardware interface to allow software developers to use their design for different neural networks. The design explores RRAM as main memory with a portion dedicated to neural network computation. Such an assumption takes a different approach than other designs, considering regular writes to the RRAM array. This DRAM-centric assumption has its drawbacks, since writing times for most RRAM cell topologies are at least one order of magnitude higher than reads. The choice of having RRAM as main memory comes from previous designs, such as DaDianNao [57], which reported very high DRAM access energy consumption.

The design proposed by Chi et al. [15], specifying the division (subarrays) inside the RRAM array, can be seen in Figure 2.11. The functional subarray has two modes: memory and computation. Additionally, buffer subarrays are directly connected to both functional and memory subarrays, and can also operate in regular memory mode.

The paper describes the circuit-level modifications and additions concerning the memory and its controller, specifically inside the functional and buffer memory subarrays, including ReLU and sigmoid units. A system-level design is also presented, highlighting the programming, compilation and execution of neural networks (CNN and MLP). Overall, the authors present a complete architecture covering the

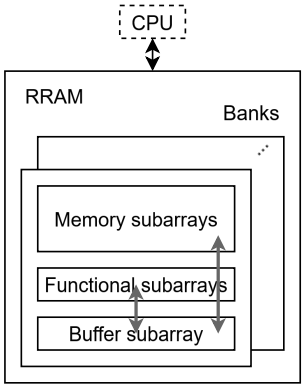


Figure 2.11: PRIME design architecture, modified from [15].

circuit design and the software integration, providing architecture transparency and ease of use. Even if the use of RRAM as a regular memory array might not be as feasible using today's technologies, the supporting digital circuitry presented and the software integration make this style of system a good alternative for further development.

2.5.3. PUMA

To set itself ahead of other CIM architectures, the Programmable Ultra-efficient Memristor-based Accelerator (PUMA) [14] comes with a specialised Instruction Set Architecture (ISA), a compiler and a detailed architecture simulator for its design.

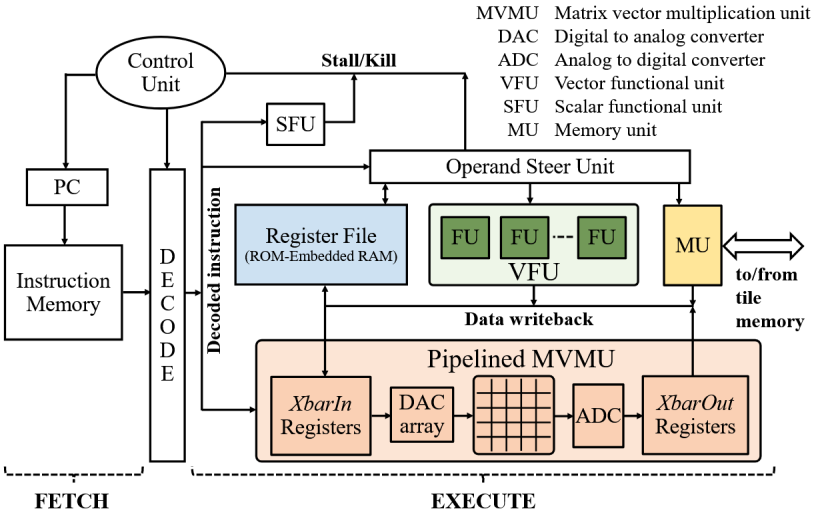


Figure 2.12: PUMA's [14] core architecture.

The architecture is organised in three levels: cores (analog crossbars, functional units, and instruction execution pipeline, shown in Figure 2.12), followed by "Tiles", which connect cores with shared memory, and nodes, which connect "Tiles" via the on-chip network, where the Core and "Tile" architectures are the main paper contributions.

PUMA's architecture adds programmability and flexibility to ISAAC's crossbar architecture [13]. Specifically, instead of focusing on CNN workloads, PUMA adds support for Long Short-Term Memory (LSTM).

PUMA's "Tile" architecture, depicted in Figure 2.13, shows how all cores are connected to a shared memory and that an instruction memory orchestrates data between "Tiles" using the Receive Buffer. At the same time, the Attribute Buffer in the shared memory is used for inter-core synchronisation by expressing the validity of the data.

A simulation is also presented as part of PUMA's contribution. The simulator runs binarised applica-

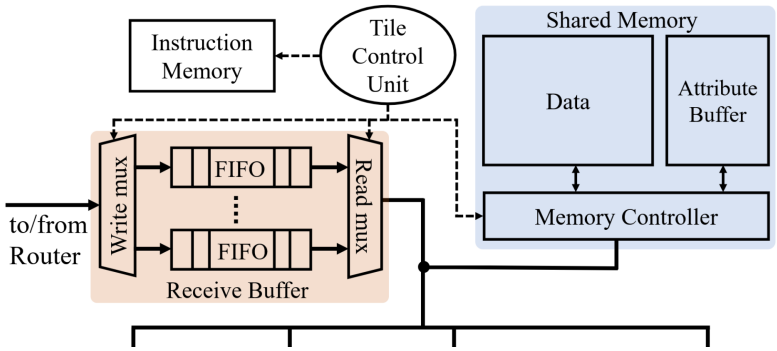


Figure 2.13: PUMA's [14] "Tile" architecture, connecting cores on the bottom.

tions compiled in PUMA's ISA and provides execution traces. Interesting for this thesis, the simulation workflow includes the following parameters:

- Area and power consumption: Using Synopsys Design Compiler [60] Verilog HDL was mapped to 45nm technology;
- Memory area, power, and latency models: Cacti 6.0 [58];
- On-chip network energy and area models: Booksim 2.0 [61] and Orion 3.0 [62];
- Chip-to-chip interconnect model similar to DaDianNao [57];
- MVM Unit power and area model from ISAAC [13].

By combining a specialised ISA, compiler, and simulator with their hierarchical architecture from cores to "Tiles" and their inter-communication, PUMA performs all the design and benchmark expected from a complete accelerator design. Its support for diverse workloads, including LSTMs in addition to CNNs and MLPs, highlights the flexibility of the architecture and compiler. Lastly, the tailored integrated simulator enables performance analysis, which is crucial for verification. Together, these contributions make PUMA a standalone CIM design ecosystem.

2.5.4. DREAM-CIM

DREAM-CIM [16] is a Digital CIM module used as the pioneer implementation for the simulation platform explored in this thesis. DREAM-CIM targets energy and area-efficient implementation of a MAC block. This block is then meant to be replicated and used in different CIM-accelerator designs.

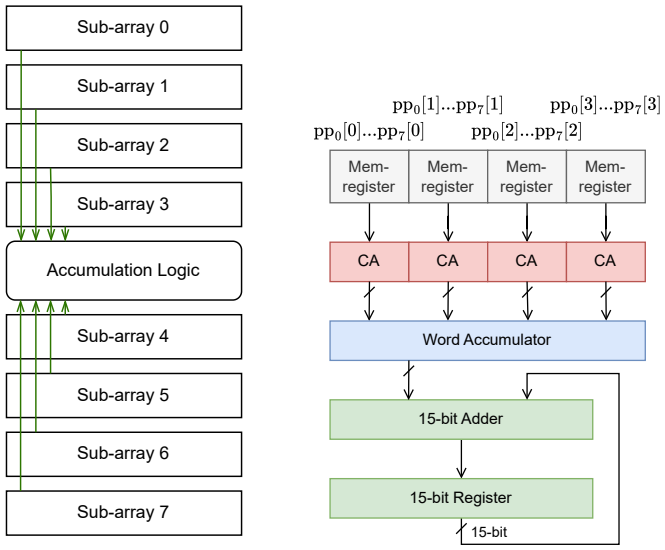
The authors point out that other digital CIM designs use adder-trees to accumulate partial products [63]. Adding adder trees leads to higher overhead in area and power, and makes the design unsuitable for small- to medium-sized applications.

The overall design can be separated into the SRAM array and the Accumulation logic. The SRAM array comprises a configuration of eight sub-arrays (banks), each with 128 columns and 16 rows. Such a configuration allows for parallel read operations between all banks. An example of how the sub-arrays are connected can be seen in Figure 2.14a. It is also possible to see the indication of Flying Bitlines (FBLs) by El Arrassi et al. [16], which are straight connections from each sub-array to the Accumulator block. There are 128 Flying bitlines per sub-array, following the number of columns.

The architecture is designed to be the most efficient at 4-bit precision for weights and inputs. For this reason, the accumulation logic supports 128 parallel read operations (one per column) distributed amongst 32 accumulation logic blocks ($128/4$).

An Accumulator block is responsible for computing the full-precision product between the input vector and the matrix stored in the sub-arrays. The accumulation can be further broken down into:

- Column accumulation: In this step, the partial product between the first bit of different input values and sequential bits of different weight values is accumulated. The bits from each input are provided sequentially, while the bits from the weights are always accessible and can be computed in parallel. Figure 2.14b shows how this step is carried out. The FBLs first provide the partial



(a) Sub-arrays definition and placement.

(b) Accumulation logic block example.

Figure 2.14: DREAM-CIM hardware components, adapted from [16].

product bits, which are stored in “mem-registers” structures. Each mem-register receives the bits from one bit position of the weight from each of the sub-arrays. In the case of 4-bit precision, four mem-registers will each be connected to a Column Accumulator (CA) unit. The output of each CA is the sum of a specific weight bit position for different weights;

- Word accumulation: The different sums are now shifted according to their weight bit position, and accumulated together. This step can be seen in Figure 2.14b;
- Bit-serial row accumulator: This last step comprises two blocks, one register to store the previous accumulated result for a single input bit and an adder to accumulate it with results from other input bits, given the necessary shifts. The output of this step leads to the full MAC output for that specific set of inputs and weights. Since the inputs are provided sequentially, this step is done over multiple cycles.

The total computation time of the system can be calculated as follows: $\texttt{precision} \times \texttt{\#(sub-array rows)} + 1$. The precision follows the sequential input feeding to the above SRAM-based crossbar array. The number of sub-array rows relates to the number of reads required per sub-array, since a read is only parallel for a single row per array. The entire system requires an additional cycle to process the last read row. Following a 4-bit precision, and the set sub-array size (128x128 with 8 banks), the computation time becomes: $4 \times 16 + 1 = 65$.

The presented architecture was simulated at circuit-level (SPICE) using GlobalFoundries 22nm CMOS technology and compared against State-of-the-Art digital CIM architectures. For 4-bit precision, as a result, a compound parameter including execution time, area and energy consumption was observed to be 38% better than adder-three architectures; it had double the execution time, but compensated by almost half of both the area and energy consumption.

Although revised in section 3.3. Three highlights in this architecture made it a suitable candidate for simulation implementation:

- Independent computation time: The amount of cycles the architecture takes to pass an input through the weights of an array is independent of the input provided, which makes the design and cycle count straightforward;
- Digital logic: Dealing with digital logic in simulation does not require any modelling of analog components and their variations;
- Familiar work: The authors could be reached to discuss the implementation details.

2.5.5. C3CIM

Just like DREAM-CIM, C3CIM also introduces a crossbar-level architecture. Biyani et al. [51] describe a constant current CIM crossbar using a new 2T1R architecture, which is also prototyped using TSMC 40nm CMOS technology. The architecture is further validated using a SNN.

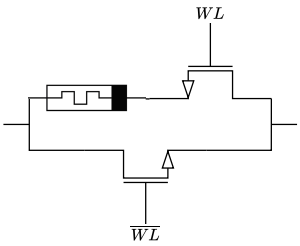


Figure 2.15: 2T1R RRAM memory cell, modified from [51].

The schematic of the 2T1R cell can be seen in Figure 2.15. The low-power benefits of this cell structure can be attributed to the complementary wordline switching and binary inputs, where only one path is active at a time. This topology leads to significant resistance only when the transistor path is selected and the transistor is in the HRS. The resistance of the pass transistor and the LRS resistance are considered negligible.

Per the design, a very low column current is provided to have the least power consumption. Therefore, the accumulation result is read as a proportional voltage output (V_{MAC}). Another benefit of using a constant column current is making the power consumed by the crossbar deterministic, tackling the non-linearity of other RRAM designs.

Given the two paths formed by the transistor pair in the RRAM cell, a drawback of this architecture is the impossibility of scaling for multi-bit inputs. However, considering the low maturity of RRAM technology for CIM, given, for example, variability, it is fair to assume 1-bit cells and input are still considerably far from being overcome.

To conclude this chapter, the foundation for digital and analog CIM architectures was revised highlighting the differences between the two. The extensive background for analog memories discussed different analog memory technologies, where RRAM was observed to be the technology in the spotlight for CIM. Later, different examples of CIM architectures were discussed, including the DREAM-CIM [16] architecture that was used as a building block for the CIM architecture framework. Building on these insights, chapter 3 provides the methodology used to define and test the CIM architecture framework.

2.6. Simulation Frameworks Used for CIM

This section provides an overview of the types of simulators and tools with potential to be used for CIM, spanning from general low-level circuit design software to specialised analytical models developed specifically for CIM. It also outlines the simulation space required to support accelerator development, highlighting areas where overlapping simulation approaches are necessary for efficient modelling, simulation, and prototyping.

Table 2.2 presents an overview of such simulators and their abstraction level. While their abstractions are explained and exemplified below, starting with the most fine-grained methods:

Circuit or Transistor level This level of simulation comprises the device physics of each transistor and its impact on the general architecture. It is generally used for analog and mixed-signal simulation of circuits, but also for larger-scale circuits. The simulation workflow starts with a circuit design coupled with a technology node, which, in turn, will dictate the physics of the circuit components used. This type of simulation is also known as SPICE (Simulation Program with Integrated Circuit Emphasis). Examples of this type of simulators include:

- Cadence [19], a well-known electronic design automation (EDA) platform that includes NETLIST generation from the circuit representation used in the SPECTRE simulation tool.

Table 2.2: Overview of current CIM simulators and their abstraction level.

Tools/Simulators	Abstraction Level
NVsim [64] ZigZag [21] CIMLoop [22] NeuroSIM [23] CIM-Explorer [24]	Analytical
Gem5 [25] GVSoC [9] Delft CIM SIM	Instruction level
NVMain [65] Mnemosene [26] DRAMSys [66] Ramulator [67]	Cycle accurate
QuestaSim [68] Verilator [69]	RTL
Synopsys [60] Cadence [19]	Circuit level

- Synopsys [60], another well-known EDA platform for designing and verifying semiconductors and electronic systems.

Register-Transfer Level (RTL) A design method describing specifically digital circuit behaviour using Hardware Description Languages (HDLs) such as VHDL and Verilog. Examples include:

- QuestaSim [68], developed by Siemens, offers a simulation platform for various HDLs, including debug and waveform visualisation;
- Verilator [69], an open-source Verilog/SystemVerilog simulator that reads, performs checks and outputs C++/SystemC code for faster simulation. This way, Verilator achieves cycle accuracy without sacrificing simulation performance for various types of simulation.

Cycle Level Simulators that offer the trade-off of faster simulation compared to RTL and Circuit Level, but still enough specificity when it comes to energy and performance of the simulated hardware. At this level, pipeline stages and ISA should be considered while simulating a processor's behaviour. SystemC[1] is a standard tool used for this class of simulators. Examples include:

- MNEMOSENE [26], a modular memristive CIM simulator written in SystemC that includes an ISA, pipeline, and CIM-tile example architecture. The simulator works with parameters that can be set by a user either in the crossbar or in the analog and digital peripheries.
- Ramulator [67], a cycle-accurate simulator, now in its second version, extensively used for performance, security, and reliability studies, supporting standard DRAM types (DDR3-5, LPDDR5, GDDR6, etc.).
- DRAMSys [66], an open-source DRAM simulator based on Transaction Level Modelling (TLM), which uses data structures for communication instead of low-level signals.
- NVMain [65], a flexible cycle-accurate simulator modelling memory requests, timing, and hierarchy for DRAM in parallel with emerging memories, suitable for performance and reliability studies in CIM architectures.

[1] SystemC is a set of C++ classes and macros that provide an event-driven simulation kernel for modelling hardware systems. It extends standard C++ with constructs to describe hardware concepts such as modules, ports, signals, and concurrent processes. SystemC has long been the industry standard for hardware/software co-design; however, even though it is embedded in C++, it introduces a concurrency model, simulation kernel, and timing semantics that make it fundamentally different from conventional C++ programming, while still not achieving the same cycle accuracy as HDLs.

Instruction-level Simulators that present a good trade-off between cycle-accurate and purely analytical simulation. They can reflect on system and hardware performance more accurately without simulating every cycle and pipeline stage as cycle-accurate and RTL simulators do. Simulators of this class use Events to transfer information and map inter-component communication and traces to present the user with the execution of the workload. Frameworks built using an instruction-level simulator can enable integration of detailed low-level models using high-level abstractions. Examples include:

- Gem5 [70, 25], an open-source and community-supported system simulator, which has become widely used and has different added features such as connection with other types of simulators. An example of this integration is with the DRAM simulator Ramulator [67]. Due to its widespread implementations, Gem5 supports different system architectures, x86, ARM®, and RISC-V.
- GVSoC [9], a system-level framework for embedded multiprocessor simulation.
- CIM architecture platform (which development is part of this thesis): Focuses on the simulation of systems built using CIM architectures.

This thesis uses simulations conducted in GVSoC and the CIM architecture platform. Therefore, their specific requirements and design workflow are discussed later.

Analytical simulators Rather than traces, analytical simulators have mathematical models to predict system behaviour. These are used explicitly in DSE to have a high-level overview of a possible design. These types of simulators are the entry point when designing a CIM accelerator. Examples include:

- ZigZag [21], used for exploring architecture and mapping strategies for DNN accelerators, emphasising uneven mappings.
- MIT's CIMLoop [22], a modelling tool that brings in flexible system specifications, energy and statistical models to streamline design.
- NeuroSIM [23], a simulator that supports integration with the popular ML platform Pytorch for performance and energy estimation.
- CIM-Explorer [24], a newly released simulator targeting BNN and TNN that uses RRAM-specific strategies. This work is also further described in a subsection below.
- NVSim [64], a memory simulator for RRAM, PCM, and STT-RAM arrays, modelling latency, energy, area, and peripheral circuits, useful for circuit-level energy and area estimation in early-stage DSE. NVSim's architecture is based on CACTI.
- CACTI [58], a memory modelling tool developed by HP Laboratories for cache and memory DSE, providing estimates of latency, energy, and area across different technology nodes and memory organisations. Still used until today, the tool has variations to also support NVMs, as the one presented above.

These simulators provide a high-level overview of possible designs, helping researchers evaluate performance, energy, and area trade-offs without requiring full cycle-accurate or RTL-level simulation.

Figure 2.16 is an overview of the mentioned simulation types and their relations. From the figure, it is possible to see how instruction-level simulators such as the CIM architecture simulator platform get energy, latency and area numbers from small-scale SPICE simulations. At the same time, analytical-level simulators have their functional verification done with RTL tools. As mentioned in their description, instruction-level simulators can communicate with cycle-level simulators, which is the case for Ramulator with Gem5, and GVSoC with DRAMSys. Ultimately, physical implementations are used to make the verification of designs, providing real-world input to all abstraction levels.

Although simulators for NVMs have been available for a significant amount of time, most focus on device- or circuit-level modelling. Equally important, however, is understanding how these memories integrate into larger systems and influence architectural behaviour, since this directly relates to their real-world applicability. To address this broader context, we introduce and compare two instruction-level and one analytical-level simulation environments.

GVSoC is chosen over another system simulator, Gem5, for an initial accelerator architecture integration to understand how event-driven simulation is performed. The reason for this choice was the

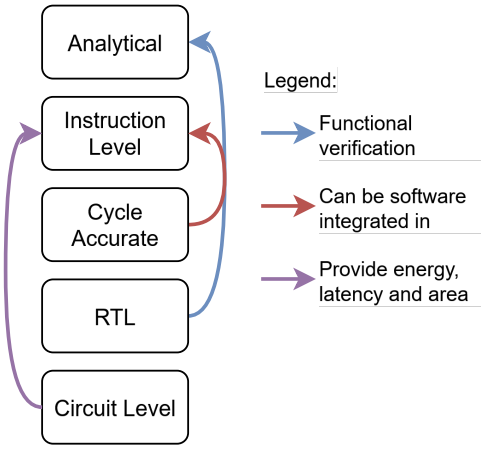


Figure 2.16: Simulation types and their relationships.

familiarity with GVSoC compared to Gem5 and the close support from the department's researchers. For this reason, GVSoC is introduced is described, highlighting how integration of CIM components can be prototyped and evaluated into a system-level simulator.

2.6.1. GVSoC SoC simulator

GVSoC [9] was, at first, developed as a simulator for the PULP project [71], supporting the development of its RISC-V architectures and simulating different SoCs from a system perspective. From its origin, it is possible to deduce how GVSoC is closely coupled with RISC-V. GVSoC's creators claim that the simulator has a medium-high timing accuracy with cycle-approximate behaviour, as cycles are not strictly tied to hardware execution, allowing faster simulation times.

In GVSoC, system components such as interconnects, processors, and memories are represented as discrete simulation objects. The platform uses Python generators to define and connect system components at the high-level architecture, while the components' implementations are done in C++. Components communicate via requests and wires, and the simulator orchestrates their interactions over simulated time, accounting for communication latency between components. Because of its close relation to RISC-V, the typical simulation workflow involves compiling a RISC-V core as the starting point, which then communicates with the declared components.

2.6.2. CIM-Explorer

As a new alternative simulator, CIM-Explorer (CIM-E), is described as an example to contrast different design philosophies and modelling approaches from analytical simulators.

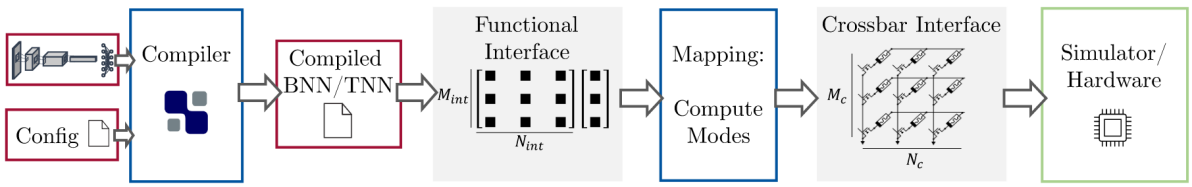


Figure 2.17: CIM-E interfaces [24].

Introduced in 2025, CIM-E [24] aims to solve different RRAM-based CIM challenges. Figure 2.17 shows the primary interfaces of the toolkit and their flow of execution, including functional and hardware interfaces. The execution starts with configuration parameters and the neural network model structure that produce the representation used for functional simulation. Figure 2.18 illustrates the modular breakdown of CIM-E for DSE, highlighting the compilation, mapping, and simulation backends. The problems that CIM-E attempts to solve are listed below:

- Map memristive device non-idealities [72]:

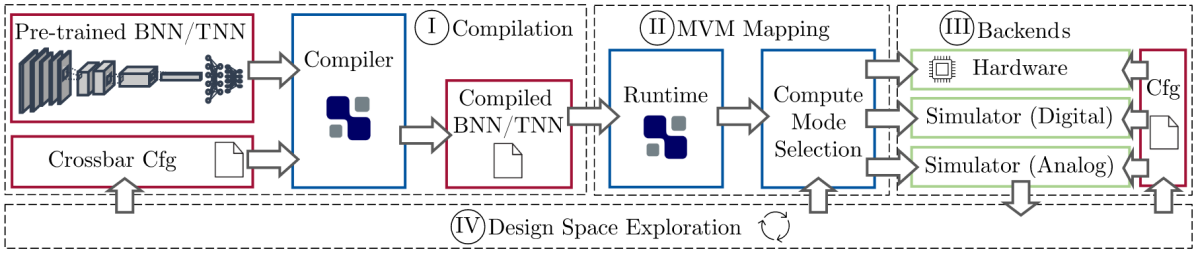


Figure 2.18: CIM-E modules [24].

  - Memristive device variability: An active research topic in fabrication and modelling [55], occurring between memristive devices and cycles of operation. Including variability in the DSE allows for an extra design parameter to be included in the design space, also opening doors to variability mitigation strategies in the context of analog CIM technologies;
  - Read-disturb: A resistance change known to occur in RRAM cells after multiple read operations [72]. It is mitigated by periodically re-writing to the RRAM cell or with dedicated control hardware, avoiding unnecessary rewrites [73];
  - Endurance: A reliability metric measured by the number of writes a RRAM cell can withstand [74]. Endurance is also linked to resistance drift, and can be considered during design and simulation [72];
  - Temperature dependency: It influences switching mechanics and retention and causes a change in device nominal resistance during the forming state of memristive devices [75]. Using temperature-dependent metrics allows designers to simulate a more realistic system and explore different alternatives during DSE.
- Perform compilation, simulation and DSE in a single platform, which was previously done in different steps and using different tools;
- Support of BNN and TNN: The non-ideality mapping capabilities allow for the simulation of more bits stored per cell and in MAC operations.

The toolkit uses Larq [76], an open-source training framework for BNNs and TNNs, and a Tensor Virtual Machine (TVM)-based compiler [2], with multi-batch support and crossbar-specific optimisations, such as maximising weight reuse. Furthermore, to streamline DSE, CIM-E links the compiler, pre-defined mapping (compute modes), and analytical techniques (hardware, digital and analog) to analyse their impact on inference accuracy. Overall, inference accuracy is used to define DSE, rather than energy efficiency or technology-specific performance. Overall, CIM-E is suitable for early design stages while offering significant room for integration with lower-level simulators.

2.6.3. CIM architecture simulation platform

CIM is an event-driven simulator created specially for CIM architectures, establishing the foundation for the performance and DSE analysis presented in the experimental results.

The platform is designed to address gaps in the current simulation landscape for CIM systems. Unlike GVSoC, which provides a general-purpose SoC framework, this simulator is tailored explicitly to CIM architectures, proposing explicit support for NVMs, data streaming, pipelining, and execution models. The experiments carried out in this thesis, and presented in chapter 4, were primarily obtained using this platform. As a target for experimentation and as a testbed for simulator features, this dual role helped define the requirements such a simulator should provide while showcasing initial use cases already implemented.

To sum up, this chapter reviews the foundations of CIM, starting with the Von Neumann architecture, its scaling limits, and the memory bottleneck. It then introduces hardware accelerators and presents neural networks as the driver of new CIM architectures. The chapter explains CIM concepts for both digital and analog memories, surveys state-of-the-art designs like ISAAC, PRIME, and PUMA. It concludes

[2] https://github.com/apache/tvm

with an overview of simulation frameworks that are used in the design of CIM architectures, which the rest of this thesis is going to tackle.

3

CIM Simulation Methodology

When it comes to the CIM architecture simulation, different alternatives can be analysed concerning the type of simulation. As described in the previous chapter, a good trade-off for the goal of this project lies in system-reflective simulation that allows for performance measuring and DSE. In this case, event-driven simulation comprises most of the simulation platform's core functionality expected from a CIM system. This entails communication and computation times.

There are two simulating platforms used during this project. The first is the SoC simulator GVSoC, and the second is the CIM architecture platform. As mentioned before, GVSoC was chosen over Gem5 for a simple accelerator system implementation to understand how simulating engine and component communication could happen in a simulating platform. This choice was given previous familiarity with GVSoC compared to Gem5. After the implementation is complete, this design served as a baseline example for the in-development Delft CIM architecture simulation platform.

Initial implementation in GVSoC The accelerator was implemented as dedicated GVSoC components, derived from the `vp::Component` class. Other system components, such as the GPU and memory, were defined in the Python constructor and connected using wire constructs. These components assisted in testing using a simple workload (an MLP model using MNIST).

Inside the C++ component space, the architecture was functionally organised to reflect the structure of a CIM accelerator. Considering the three layers of the MLP model, this leads to three main components, separated by purely functional ReLU modules.

Simulation at this stage aimed to keep the system as simplified as possible, intending to implement additional constructs and add granularity later. Therefore, no dedicated control component was used; the compiled RISC-V code acted as the system CPU to control simulation start and finish. Data streaming was also implemented here, using 4-byte communication packages. Each of the components handles the request coming from the CPU and other components, and by doing that, it can also reply to such requests with a delay. This feature was used to get the required architecture-dependent cycle delays.

After the CPU starts execution (acting as the controller), the subsequent actions follow linearly: input data is loaded 4 bytes at a time for computation within a group of MAC blocks, the computation is performed, the output is passed to the ReLU, and the ReLU then sends the modified data to the next MAC block group for the second layer, and so on. Quantisation was applied only functionally within each MAC block component.

No pipeline was assumed; instead, the simulation relies on large enough memories in each group of MAC blocks. At a lower level, the simulation follows the unbounded problem principle that will be further described in subsection 3.3.7: each crossbar has a fixed size, and the number of crossbars depends on the loaded matrix. There was also no component-level distinction within the MAC block groups to maintain simulation simplicity. While this lack of granularity may lead to non-ideal communication time and energy being accounted for, it provided a fast-to-implement working system.

GVSoC includes built-in models for cycle counting and energy estimation at the component level. At this stage of the work, the expected finishing time of each component was calculated internally before returning results. Energy data was also collected, but only for learning purposes; it did not represent the expected energy spent in a real implementation.

Overall, describing a CIM architecture in GVSoC required a stretch on the simulator's capabilities. Instead of simulating a small-scale SoC, GVSoC has to be configured so that the most relevant parts reside in the C++ component space and their communication.

3.1. Simulator definition

The platform is built around an event engine for cycle simulation, where events can be scheduled using C++ lambda functions. A class diagram exemplifying this simulation mechanism can be seen in Figure 3.1. Such events allow fine-grained control of simulated time by enabling concurrent activations across pipeline stages. To increase modularity, each component can schedule events within its scope, ensuring that cycle counting is performed directly on simulated time rather than by assigning fixed, estimated cycle counts to operations.

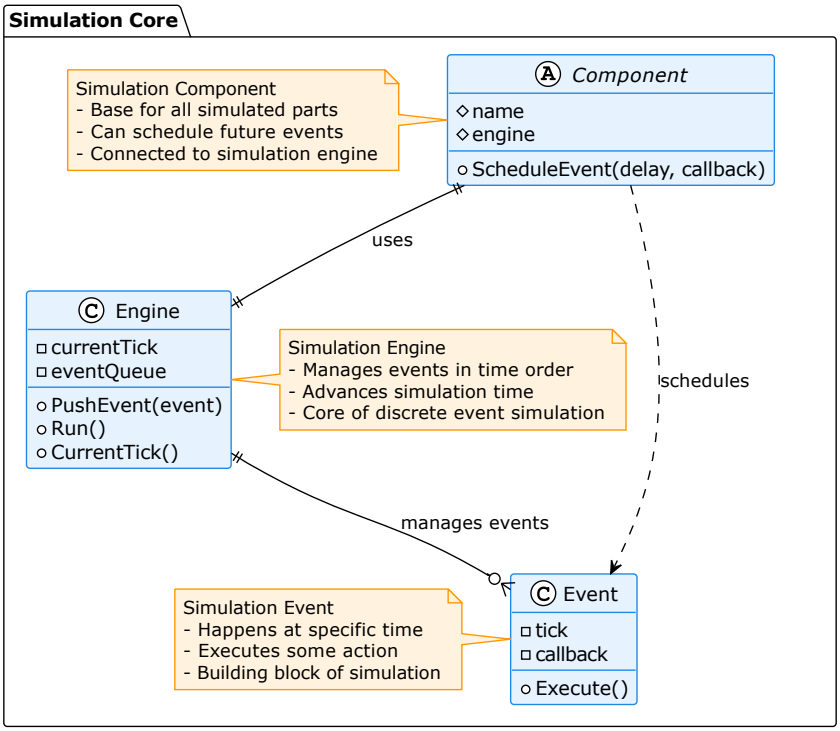


Figure 3.1: Simplified class diagram of the simulation core used in the CIM simulation platform.

Apart from event scheduling, the simulator incorporates a request–response mechanism for component communication, a common approach also used in GVSoC. A class diagram exemplifying the different pieces of this communication can be seen in Figure 3.2. Requests act as lightweight messages that allow one component to trigger behaviour in another, or send data that needs to be handled. This design is particularly well-suited for modelling multi-component architectures, where computation is distributed across several interacting modules, a key feature of CIM systems.

To stand out amongst different system simulators, the CIM architecture simulation platform would enable the integration of NVM physical switching models as standard components to provide a more accurate simulation of CIM architectures. Moreover, leveraging the speed and power of a C++ framework, while providing a platform for the comparison of different memories for CIM.

The definition, implementation and testing of the platform in this thesis meant:

- Define the simulation engine;

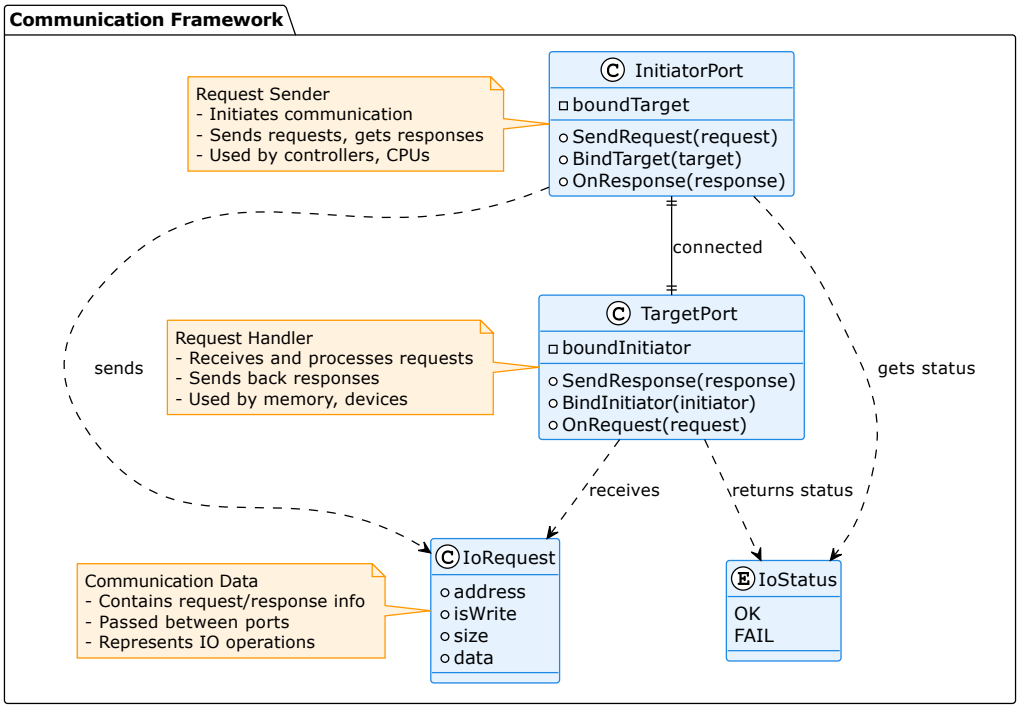


Figure 3.2: Simplified class diagram of the communication framework used in the CIM simulation platform.

- Standardise how hardware components and their communication are defined, and make it the initial standard for the simulator;
- Test the event engine's intended functionalities;
- Analyse the ease of use of the simulator, considering system definition based on an example architecture, DREAM-CIM [16], and workloads.

3.2. Simulated system organization

The first step to simulate a system including a CIM accelerator was to define the high-level component distribution. This division is meant to separate the MAC blocks (referred to here as tiles) from the controller, memory and extra hardware that take care of other digital computation tasks.

3.2.1. General system architecture

A simple architecture was chosen to streamline development when considering different architectures with three main components, following the idea that analog accelerators will be mainly used for inference (where the weights are already defined) and not training. Therefore, weights will be preloaded into the crossbar array and not count towards the computation time of the accelerator.

The exemplified system architecture can be seen in Figure 3.3, where dotted arrows exemplify flow during setup time and full arrows exemplify flow during normal execution. A breakdown of each component and the corners cut in their development are listed next:

Controller is the simulation entry point, either with a modelled behaviour, such as a batch schedule, or a simple entry point that executes the actions required to start inference. These are: Load the input into local memory, and send an interrupt signal to the accelerator component to start computing.

Memory is a Middle storage unit that acts like a double-sided buffer between the controller and the accumulator. An abstraction was made here, where in a hardware architecture, extra intermediary components should intermediate the communication between memory and accumulator, such as a Direct Memory Access (DMA).

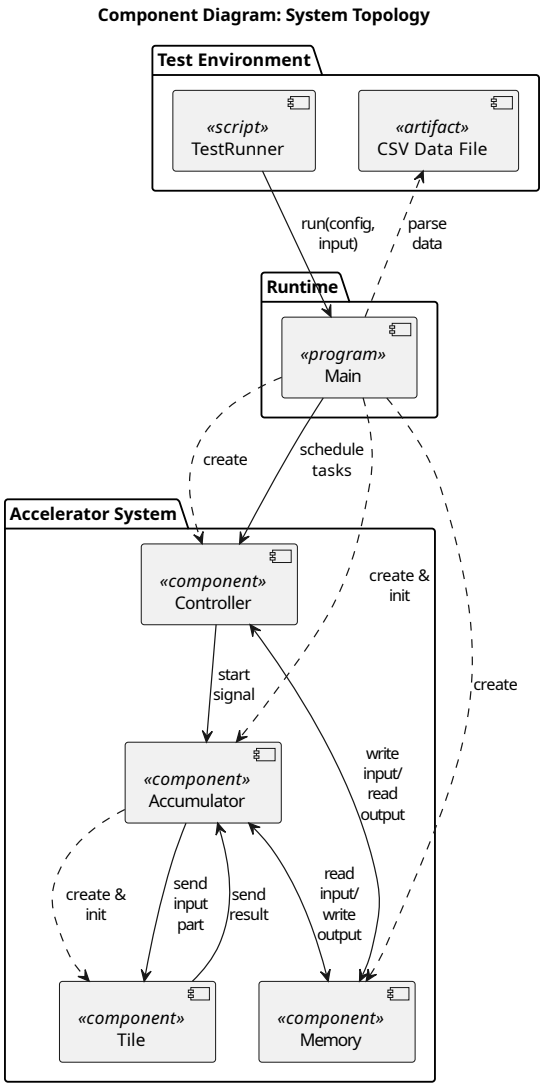


Figure 3.3: General component abstraction diagram.

Accumulator is where the simulation parameters must be well defined. It loads the weight matrix on program initialisation, receives an interrupt request from the controller, can read the input array from the memory, and writes the computed result back to memory. The chosen way to get the data back from the accumulator by the controller would be a scheduled time to read the data, ensuring that the data would be available. This method comprises a systolic array-like execution, where the computation is expected to be done in a fixed number of cycles, allowing communication to run smoothly and avoid race conditions. Another option would be using an interrupt response, which would be a stretch for real hardware execution due to the granularity of component definition at this stage, or polling from the controller side, which would add an extra level of complexity for this stage of development.

Tile is where the micro-architecture is included, with its sub-components and cycle count logic to simulate hardware. An abstraction that contains a memory crossbar and the supporting periphery to perform a computation.

The CPU simulation can also be managed with the rest of the defined system topology. For example, this simulated CPU would be a RISC-V core that offloads work to an accelerator component, with its local controller, memory and accumulator. A CPU implementation is not considered at this stage, as the entire system simulation would make the analysis much more surface-level and deviate from the goal of showing accelerator performance focused on the tile micro-architecture.

3.2.2. Streaming

The concept of streaming inside a CIM architecture comes from a perspective of data movement. Streaming is the alternative to intermediate large storage between computation nodes, allowing data to continue to the computing path and be minimally stored (buffered) until the output is reached. Some key characteristics of streaming in CIM are:

- Inputs are serially injected into computing units (tiles) rather than stored locally;
- Intermediate data generation may not be explicitly stored, using communication channel delays to pass along hardware pipelines.

As of today, the most famous CIM architectures implement some streaming architecture/pipeline to handle workloads efficiently and with smaller area overhead due to smaller buffer units. Two examples of these architectures are ISAAC [13] and PUMA [14].

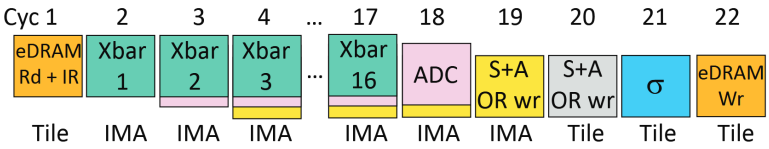


Figure 3.4: Example of one operation in a CNN layer flowing through its pipeline in ISAAC [13].

ISAAC implements pipelining by utilising the eDRAM buffer in each "Tile" (a level of abstraction defined by the authors that holds multiple MAC blocks). This buffer does not need to store all the output from one layer before allowing the hardware to move to the next layer. This flow allows for parallel execution and, therefore, pipelining. Figure 3.4 shows how a layer traverses through the pipeline, passing through the different components, while also considering the number of cycles required by these components and communication between them. More operations can continue after the presented pipeline, for example, the `max-pool` that would happen only every few iterations of this whole pipeline could happen in cycles 23 to 25 (considering reading, operation and writing).

PUMA [14] implements streaming at a core (a crossbar and its periphery, called a core by the authors because of its instruction decoding capabilities) and multi-core level by introducing storage between components. At the core level, these include the Register File and the Memory Unit. At the multi-core level, the architecture also implements Shared Memory and a Receive Buffer, both allowing data to be distributed amongst cores and core clusters, avoiding race conditions and using the bandwidth present to "hide" computation time. As the architecture has an ISA and three pipeline stages (fetch, decode and execute), streaming is realised through constant dataflow from one component to another.

Streaming indicates that the accelerator architecture can be effectively used and tested for different workloads. However, buffers, FIFO queues, and local tile memories must be realised at the accelerator level. For this thesis, and to match the topic of simulating different architectures with variable parameters, streaming is considered at the tile and inter-tile level, where a tile refers to one crossbar with its accumulation logic. Streaming can be realised across multiple tiles by leveraging assumed communication delays and buffering between tiles and the higher-level inter-tile accumulation logic.

3.2.3. Quantisation

A quantised pre-trained MNIST dataset was used to test the implemented example architecture, more specifically, a three-layer inference of a simple MLP model, where quantisation was applied to each layer. For testing, PyTorch [77] was used to export the weight matrix for each layer together with their quantisation parameters.

Quantisation [78] refers to a scheme used to change the float data type of the parameters for more straightforward integer computation. Quantisation facilitates the design of CIM architectures, since they focus primarily on the MAC operation of MVMs.

$$r = S(q - Z) \tag{3.1}$$

A quantisation scheme comprises parameters presented in Equation 3.1, where r stands for "real value",

q for “quantised value”, S for “scale” is a positive real number, and Z for “zero-point”. During matrix multiplication, the following composition of different scaling values occurs [78]:

$$q_{out}^{(i,k)} = Z_{out} + M \sum_{j=1}^{N} (q_{in}^{(i,j)} - Z_{in})(q_{weight}^{(j,k)} - Z_{weight}), \text{ where: } M = \frac{S_{weight} S_{in}}{S_{out}} \tag{3.2}$$

During calculation, it is possible to see a single floating-point operation using the same scale parameter M, followed by a zero-point addition for each quantised output. Meanwhile, the input and weight matrices already undergo zero-point subtraction.

In CIM accelerators and regular hardware, quantisation is also used to streamline the execution of different models, reducing latency due to limited storage or computing power.

3.3. Target architecture implementation

As the pioneer implementation to be simulated into the CIM architecture simulation platform, DREAM-CIM [16], was chosen due to the following factors:

- The computation time per designed tile is workload independent. This means that there was only one logical path in the design, and independently of the computation taking a few or many inputs at once, the output would be produced in the same amount of time, which simplifies troubleshooting;
- Separation from NVM-specific details, enabling parallel system implementation of NVM metrics, which, in turn, facilitates the study of CIM architecture integration into a complete system with reduced complexity;
- The familiarity with the development team and the origin of the design being published work by a researcher from the same department, which reinforces its reliability and fast implementability.

The proposed micro-architecture (with a detailed description in section 2.5) does not constitute a whole accelerator system architecture, but rather a building block of this system: a MAC unit, referred to as a tile. An overview of how micro- and macro-architectures come together in CIM-design can be seen back in Figure 3.3.

To simulate the DREAM-CIM implementation for inputs and weights with 4-bit precision, first, the individual containers were declared following the described boolean architecture previously shown in Figure 2.14b:

- Crossbar: A vector container of 128 rows of 128 bits;
- Input register: A vector container of 128 rows and columns equal to the number of bits, allowing for simple sequential access;
- Mem-register: A vector container with space for eight bits, one from each bank subdivision of the crossbar. This container repeats for each precision bit of the weights. Then, for all accelerator blocks that compute the accumulation in parallel.

To maintain computation integrity, individual integer registers were used for calculation purposes:

- Column accumulator: Stores the accumulated partial products per Mem-registers, for each weight precision and sub-array row;
- Word accumulator: Stores the accumulated column results per weight bit-precision, for one sub-array row;
- Row accumulator: Stores the accumulated word results per sub-array row;
- Serial accumulator: Stores the accumulated row results over different sequential input bits, recomputing the above registers.

Furthermore, each of the above components has one extra top-level dimension to reflect the 32 parallel accumulator blocks in the architecture.

The architecture was also made flexible to different bit-precisions, but at the cost of changing the size of the accumulator block components. A list of parameters used in the simulation can be seen in Table 3.1.

Table 3.1: Parameters considered in DREAM-CIM.

Parameter	Value
#array rows	128
#array columns	128
#banks (sub-arrays)	8
Sequential row reads	$\frac{\#\text{array rows}}{\#\text{banks}}$
Input precision	4
Weight precision	4
Output features	$\frac{\#\text{array columns}}{\text{weight precision}}$

Changing the precision from 4-bit to 8-bit, for example, decreases the number of parallel accumulation blocks in the architecture and consequently the number of output features. However, the number of mem-registers and Column Accumulators increases at the same time.

3.3.1. Signed arithmetic

Signed arithmetic is essential when considering workloads for the DREAM-CIM tile architecture, especially when different bit precisions are supported. While the original design does not implement signed arithmetic, the architecture is extended to support it in this work. Specifically, two changes are introduced: one to handle signed synaptic weights and another to handle signed inputs, enabling fully signed computations.

As also used in ISAAC [13], to support signed weights, the value resulting from the Most Significant Bit (MSB) of the weight is multiplied by $-2^{\text{bit-precision}}$. Similarly, for signed inputs, considering the inputs are provided sequentially, it is enough to perform a shift and subtraction, instead of shift and add for the last and MSB of the input. This scheme assumes 2's complement, where the MSB is the sign bit of the number representation. The idea of this method can be further specified [79], when dealing with this multiplication algorithm, to allow for sign bit representation, it suffices to make 2's complement conversion ($\sim$result + 1) for the computed column sum corresponding to the weight sign bit (MSB) after normal unsigned accumulation.

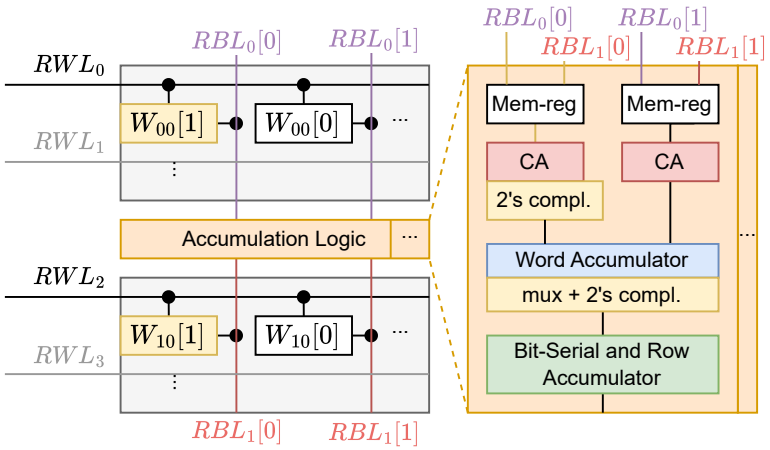


Figure 3.5: Representation of handling synaptic weight sign-bit arithmetic in hardware, modified from [16].

Handling 2's complement signed numbers in the pipeline is exemplified in Figure 3.5. On the left side of the figure, it is possible to see how the multiplication of the sign-bit from different synaptic weight values is accumulated per column in its path. Therefore, only one accumulation path requires additional logic to handle 2's complement. Assuming such a change does not require any timing change, the accumulation pipeline can continue as planned.

3.3.2. Simulated metrics

The simulator framework in this work was built to provide metrics that form the basis for analysing CIM architectures. These include the number of active tiles, tile utilisation and execution cycle count. Together, these metrics give an idea of workload parallelism and accelerator size to be considered further in accelerator design based on performance costs. These metrics also allow for fundamental design trade-offs such as resource allocation and throughput balancing.

Energy consumption is not included among the reported metrics in this work. While it is one of the primary motivations behind CIM, in the current development state of the simulator, energy results would not provide reliable or meaningful insights. Energy was chosen not to be included in the focus due to the lack of validation against hardware or established models. Excluding speculative energy data makes the reported results consistent, focused and relevant to the design decisions that can be made at this stage. Future simulator versions are expected to integrate energy estimation to enable a complete evaluation of CIM systems.

3.3.3. Pipeline

The design has a parallel and serial accumulation path. The parallel comprises most sub-components: Mem-registers, Column Accumulators, and Word Accumulators. The Word Accumulation is a parallel part because it is the last step within one accumulator block, where all previously read bits are accumulated together. The parallel part here is the one to take care of all the bits of the weight in parallel, for a single bit of the input. Consequently, the serial part repeats this computation for each input bit provided serially.

The other part of the accumulation logic consists of the bit-serial and row accumulator, which shifts the result from the word accumulator and, after each cycle, adds to itself until the output is computed. Given these two distinct pipeline stages, the total cycle count of the whole micro-architecture can be summarised as the equation:

$$\text{input precision} \times \frac{\#\text{rows}}{\#\text{blocks}} + 1 \tag{3.3}$$

For the default parameters presented by Arrasi et al. [16], 128×128 array, 4-bit precision and eight blocks, this leads to a total cycle count of 65 per tile. One extra cycle is needed to finish accumulating after the result of the last input bit goes through the accumulation logic.

3.3.4. Tiling organization

To handle a workload, an accelerator system generally comprises multiple tiles, CIM blocks that carry the same micro-architecture. In hardware, these blocks can be connected using different interconnection topologies. The study of different topologies is not in the scope of this thesis. However, it is worth mentioning that the intercommunication timing and topology between these tiles constitute a design parameter to consider when designing an accelerator.

Neural networks are very often constituted of synaptic weights of multiple sizes. Bigger synaptic weight matrices have to be distributed across various tiles. For example, each tile can be loaded with a part of the synaptic weight matrix. An accumulator should then take care of the element-wise sum or concatenate each tile's possible output. The most straightforward way to do this is by using the concept of matrix tiling.

Tiling in the context of matrix multiplication is a widely used optimisation technique that improves performance by enhancing memory access efficiency. The core idea is to divide large matrices into smaller sub-blocks, or "tiles" that can be loaded faster, using lower-latency memory (such as shared memory in GPUs) before computation begins. This setup reduces the number of slow global memory accesses and enables better use of parallel hardware resources. In CUDA programming, tiling is fundamental to achieving high performance, as it allows threads within a block to load and process data while minimising redundant memory accesses cooperatively. It is particularly effective when paired with coalesced memory accesses and loop unrolling techniques, which can significantly speed up matrix operations [80] by processing parallel "chunks" of data.

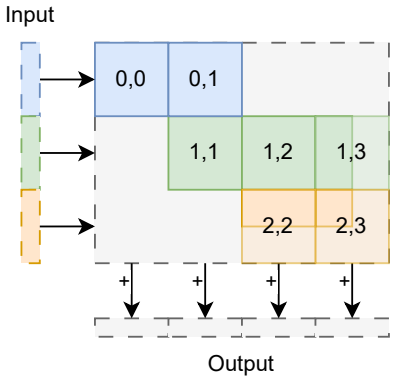


Figure 3.6: Example of tiled matrix-vector multiplication.

Regarding the chosen computing tile organisation in the target architecture for the simulated platform, the tiling concept can be seen as each physical crossbar holding a matrix section. Figure 3.6 depicts an example of a tiled MVM where the same part of the input vector is written to all the tiles in that row, and the resulting multiplication of a column is added together to form the whole output for a specific part of the output. For this reason, keeping track of tile positions inside the accumulator is essential to performing the proper operation (either element-wise accumulation or simple concatenation of the results from each tile).

To simulate these additions and connections, additional hardware has to be assumed when simulating a general architecture that allows matrices larger than a single crossbar size. This hardware is also considered to have an area and delay overhead, which are presented and discussed further.

3.3.5. Utilisation

When speaking about each crossbar storing values from a weight matrix, the matrix size often does not precisely match the crossbars' total size. For example, considering a matrix of size 100×100, and crossbars that hold 64×32 values, there cannot be an arrangement of 1×3 crossbars, since there will be leftover values; the arrangement then needs to be at least 2×4. Considering the example above, the maximum matrix size the arrangement can hold is a matrix of size 128×128. Figure 3.6 also shows an example of imperfect utilisation, where the bottom row and the right column are not fully populated. A formula for the overall utilisation is deduced and calculated as follows:

$$\mathrm{U} = \frac{\text{Problem size}}{\text{Allocated tile space}} = \frac{M \cdot N}{\left\lceil \frac{M}{T_M} \right\rceil \cdot \left\lceil \frac{N}{T_N} \right\rceil \cdot T_M \cdot T_N} \tag{3.4}$$

Where M is the number of rows in the matrix, N is the number of columns, T_M is the number of rows per tile (tile height), and T_N is the number of columns per tile (tile width). The Ceil function assists in calculating the total number of needed tiles in both directions.

In hardware, the dimension mismatch is handled with zero-padding, where both the input and the affected tiles have to be zero-padded, leading to results that can be ignored. Following Equation 3.4, it is possible to see that utilisation is its maximum (100%) if $M \mod T_M = 0$ and $N \mod T_N = 0$, or in other words, when the matrix dimensions are exact multiples of tile size.

There are two ways to consider how the workload will fit into the simulated architecture. The first, and closer to real hardware implementations, is to consider that the hardware is constrained to a fixed size, namely a bounded problem, for the better or worse. For better when the input fits nicely inside the fixed number of tiles, or for the worse when time and energy are wasted to compute a very small matrix multiplication. The second is to consider that the hardware at this stage can be analysed in simulation, depending on the workload, and find an optimal size for the considered workloads. Both approaches have pros and cons and are used in different stages of development.

3.3.6. Bounded problem

The number of tiles in hardware is always fixed, even when not all tiles are used. The fixed number of tiles defines a bounded problem, where the same number of tiles can be assumed for every workload. This topology leads to a different type of study to maximise tile utilisation by either splitting different problems inside the same tile, or using a fraction of the tiles for one problem and another fraction for another. The first leverages the most space and parallel capabilities, but does so at the cost of added hardware to keep track and separate the results adequately. The second, simpler division, has lower utilisation and still needs logic to track where one multiplication ends and where another multiplication starts, keeping track of the time spent in each computation.

The bounded problem relies on a fixed architecture, where the main research topics would be: How inter-tile communication is conducted, how utilisation can be maximised, synchronisation of different matrix multiplication operations, and overall, an exploration of programming methods.

The consideration of a bounded problem falls under the assumption that the hardware components are already defined for the entire accelerator system, which is beyond the scope of this thesis. For bigger inputs, a somewhat bounded problem can be used to analyse how many repeated cycles might be needed to complete a layer.

3.3.7. Unbounded problem

An unbounded problem does not consider a fixed amount of tiles; instead, it allows for architecture DSE by reporting on the area and delay of the architecture, considering simple system building blocks. An example would be the study of how many tiles are required for a specific workload, or how different tile architectures impact the overall performance in terms of cycles.

An unbounded problem is the primary consideration for the remainder of this project, where a general accumulation architecture is considered to conduct the study on the number of tiles and delay of different workloads to demonstrate the accuracy and flexibility of the simulation platform.

3.3.8. Generalisation to other architectures

While DREAM-CIM was selected as the reference implementation, the methodology developed here is portable across different CIM architectures. The following options enable this portability:

- Configurable tile design: Key architectural parameters, such as tile size, number of parallel reads and precision, can be adjusted to align their timing with other CIM implementations. Furthermore, because of the implementation of a modular accumulation logic, such blocks can be changed to match the delay of different architectures;
- Device abstraction: Alternatively, the entire tile component can be swapped with a simple change of communication ports, while other architecture components stay the same;
- Workload mapping strategy: Even if simple, tiling mechanisms are not tied to the design but to the parameterisable crossbar size, making the simulator workloads portable across different architectural organisations to be implemented.

  Therefore, although DREAM-CIM provides the initial test case, the approach offers a foundation for systematically exploring and comparing a broad spectrum of CIM architectures.

3.4. Tunable parameters

To perform a fair comparison between different workloads and exemplify the framework's capabilities, tunable architectural parameters were defined among the hardware capabilities of DREAM-CIM [16]. The architecture is presented with baseline parameters depicted in Table 3.1.

- **Number of rows**: Allows more parallelism for bigger inputs and longer weight matrices. Making such a parameter programmable allows for design exploration of other state-of-the-art memory sizes and their benefits. Together with the number of banks, it defines how many sequential row AND operations will need to be performed following the relation $\frac{\#\text{rows}}{\#\text{banks}}$;
- **Number of columns**: Similarly to array rows, the number of columns allows for more extensive exploration. However, since the bits of a weight value are arranged over multiple columns, in-

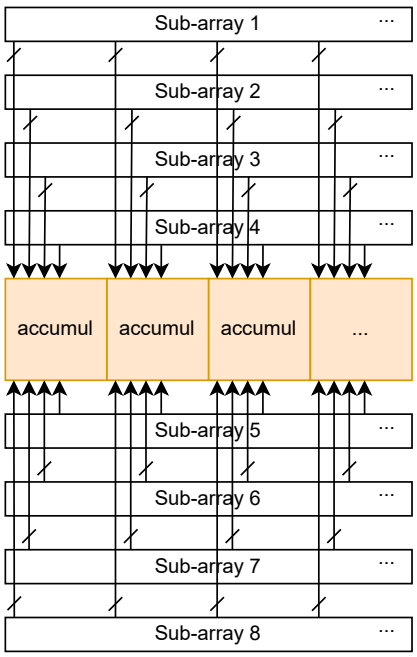


Figure 3.7: Flying bitlines connected to multiple accelerator blocks, promoting parallel computation, modified from [16].

creasing the number of columns increases possible outputs per tile. The columns of the array define the number of parallel accumulator blocks present in the architecture. The number of accumulator blocks follows the logic in the previous chapter, which is also shown in Figure 3.7. The topology dictates that the more columns the sub-arrays have, the more accumulator blocks and FBLs will be necessary to compute the parallel output, while maintaining the bit-precision;

- **Number of banks**: Allows for more parallelism due to increased area and data connections, since the number of rows per bank directly correlates with the number of cycles to produce the output. The number of banks is tightly coupled with the number of rows, so these two parameters should often be changed simultaneously to avoid influencing the computation cycle count;
- **Input bit-precision**: Allows for more flexibility concerning datatypes. Which, in turn, facilitates the use of quantised workloads since quantisation is usually conducted at 8- or 16-bit intervals for standard neural network models. The input bit-precision directly influences the number of cycles needed to accumulate the output, since each input bit is provided serially;
- **Weight bit-precision**: To allow for extra configurability, the bit precision of the weights can be changed independently. Although this separation was not often observed in current literature, its area impact on the specific target architecture for this project served as motivation to have two definitions. The effect on the area comes from each mem-register storing the 1-bit results from all sub-arrays for one bit-position. Therefore, for a bit-precision of 16, 16 mem-registers should be used. Apart from the changes in the word accumulator and serial adder to support more bits.

As seen from the overview above, each change in hardware parameter also changes the hardware requirements. Therefore, configuration packages were chosen to conduct a comparison between different defined system topologies. The number of rows and columns was varied between 64-256, considering that this was the range observed in literature from which most CIM crossbar arrays lie. The number of banks was only changed between 4-8 to not deviate from DREAM-CIM's design. Bit-precision was always changed as a single parameter to keep consistency between the chosen configurations, and leaving the weight differentiation between input and weight precisions for further research. The bit-precision range is then 4-16 bits, to exemplify the most observed quantised ranges in literature [81]. The configuration packages are as follows:

1. **Baseline**: This configuration carries the baseline array size, input/weight bit-precisions, and number of banks as shown in Table 3.1;

2. **Smaller array**: As the name suggests, this configuration decreases the array size, leading to more tiles needed for a computation but higher utilisation. Such topology was chosen to test the simulator's capabilities to change the hardware inside the tile, making it support not only a smaller array, but also the matching sub-components inside the tile that communicate together to complete the MAC operation. As an attempt to follow hardware constraints, from the DREAM-CIM architecture, the number of banks was also decreased by the same factor as the number of rows in the crossbar array. This choice follows from a standpoint of making the whole tile smaller without changing its density, including the accumulation logic. The ratio between read bit lines and peripheral circuitry is then maintained, following a realistic scaling of the circuit, while also considering the energy-area metrics regarded in the baseline design. If the same number of banks were kept, the number of cycles to execute the tile would also reduce, following Equation 3.3;
3. **Increased precision**: Changing the precision is crucial to allow for reconfigurability, and the possibility of using the most different neural network models. In this configuration, the precision was quadrupled, 16-bit precision for input and weights, which is still half compared to the current 32-bit arithmetic in modern computers. The choice of only changing the precision for both input and weights stems from its impact on capacity from the baseline architecture, hopefully leading to a promising architecture towards a balanced configuration;
4. **Balanced**: After going through different significant changes from the baseline parameters, this configuration aims to be a balanced alternative for typical neural network workloads by supporting 8-bit input and weight precisions. Apart from that, the size of the crossbar array is also changed by doubling only the number of columns. As a result, this configuration will double the amount of RBL, ultimately maintaining the same number of tiles as the baseline parameter configuration for the same workload. There will also be double the number of parallel accumulation blocks inside the tile without increasing the size of the subcomponents inside them.

	Baseline	Smaller array	Increased precision	Balanced
Number of rows	128	64	128	128
Number of columns	128	64	128	256
Number of banks	8	4	8	8
Bit precision	4	4	16	8

Table 3.2: Overview of chosen configurations and their parameters

Following the four defined configurations, to demonstrate the simulator's capabilities, layers from different workloads were also chosen.

3.5. Simulated workloads

As presented in section 2.4, CIM architectures are designed to run neural networks. This choice comes from a demand for fast, small and energy-efficient inference through multiple computing platforms and embedded devices. As for the time this thesis is being written, on various platforms, inference with these characteristics needs a micro-architecture that streamlines general matrix-matrix multiplication (GEMM). Multiple types and models could benefit from CIM architectures in neural networks. To be mapped into hardware, convolutional layers from CNNs must be flattened by undergoing the $im2col$ algorithmic transformation. The following section further describes this algorithm.

3.5.1. Mapping CNN into GEMM

Hardware accelerators are commonly built to perform GEMM, a memory-intensive task. Furthermore, although most neural networks include fully connected layers, convolution is a significant and influential part of current neural networks. To bring the benefit of parallelism and to convolution calculations, the $im2col$ (image block to column) algorithm flattens each convolutional window, vectorising the convolution algorithm to be treated as GEMM. This process increases parallelism at the cost of possibly increasing memory usage by making copies of the input.

This process of lowering convolution into matrix multiplication is exemplified in Figure 3.8. It is possible to see how a block of the input is vectorised according to the size of the convolution kernel. Furthermore, data duplication is present depending on the kernel stride, which guarantees efficient vector

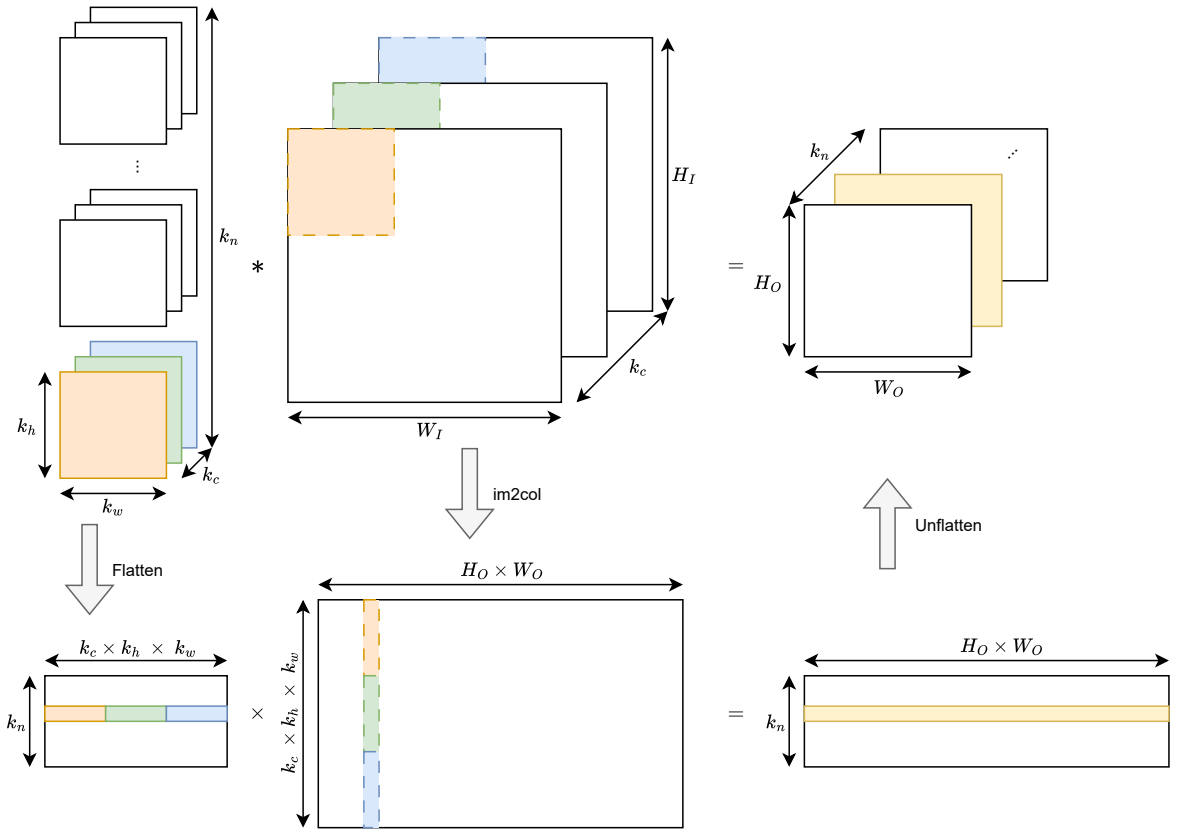


Figure 3.8: Im2col algorithm and kernel flattening.

computation by increasing the amount of data stored. The weights of the convolution layer are equally vectorised, leading to a simple GEMM. As a last step, this output should be reshaped back to its expected format so that the output can proceed to the next step in the model. Since the weight kernels are constant for all inputs during inference, these are flattened and programmed into the crossbar, avoiding data rewrite.

To transform a multi-channel input tensor of shape: $k_c \times H_I \times W_I$, into a matrix, the following steps are taken:

1. The input tensor is zero-padded with the specified amount p in all spatial dimensions;
2. For each sliding window position defined by the kernel size (k_h, k_w) and stride s, a patch is extracted and flattened into a column vector of size $k_c \times k_h \times k_w$;
3. The flattened patches are arranged as columns of a matrix with dimensions $(k_c \times k_h \times k_w) \times (H_O \times W_O)$, where: $H_O = \frac{H_I + 2 \times p - k_h}{s} + 1$ and $W_O = \frac{W_I + 2 \times p - k_w}{s} + 1$.

While $im2col$ has to be employed for each input, weights are stored in memory already flattened. The process of flattening these weights involves just one extra top-level variable, the number of filters (or output channels) k_n, leading to a matrix with dimensions $k_n \times (k_c \times k_h \times k_w)$. As it is now intuitive to see, the output matrix will then have dimensions $K_n \times (H_O \times W_O)$ before it is unflattened to its proper format $k_n \times H_O \times W_O$.

3.5.2. Choice of ANN models

To exemplify the simulator's functionality and introduce different possible workloads, a simple MLP, convolutional layers of CNN models and a layer from a transformer model are simulated, considering the different presented datasets that each model uses.

Layers from different ANN models were then chosen to perform a comparison considering different

workloads, exemplifying the simulator's capabilities. Even though the simple MLP model was run as a whole to test the simulator, a specific layer was chosen for consistency in the results. At the same time, only particular layers from the CNN and Transformer models were simulated. Each model was exported whole using PyTorch [77] in the ONNX format, allowing it to be visualised using Netron [82]. And layers could be chosen based on size and relevance to the comparison.

The goal of the comparison between the workloads is to observe how the system defined in the simulator behaves for different layer sizes for both fully-connected layers and convolutional layers.

Simple MLP Considering the possibilities, the first and most obvious contender to start testing the simulation built is a simple MLP model, which can be built and trained locally using MNIST. To assist the training of the model, PyTorch [77] was used, where it was also simple to access the MNIST dataset. The chosen MLP model consists of three fully connected layers, separated by ReLU activation functions. The model expects a typical MNIST input, with 784 values (28×28), the first layer with an output of 512, the second layer with an output of 32, and finally, a 10-value output prediction, all for quantised input and weights. Where the first layer with input size of 784 and output size of 512 was chosen for simulation.

Table 3.3: Parameters and their meanings, Layer 1 of LeNet.

Parameter	Value	Meaning
Output channels	6	Amount of feature maps this layer will produce
Input channels	1	Amount of feature maps that the layer expects, one since the image is greyscale
Kernel size	(5, 5)	Height and width of each of the filters, captures spatial features, such as edges
Input size	(1, 28, 28)	A single feature with the whole image as the spatial size
Output size	(6, 24, 24)	Multiple channels with reduced size due to filtering

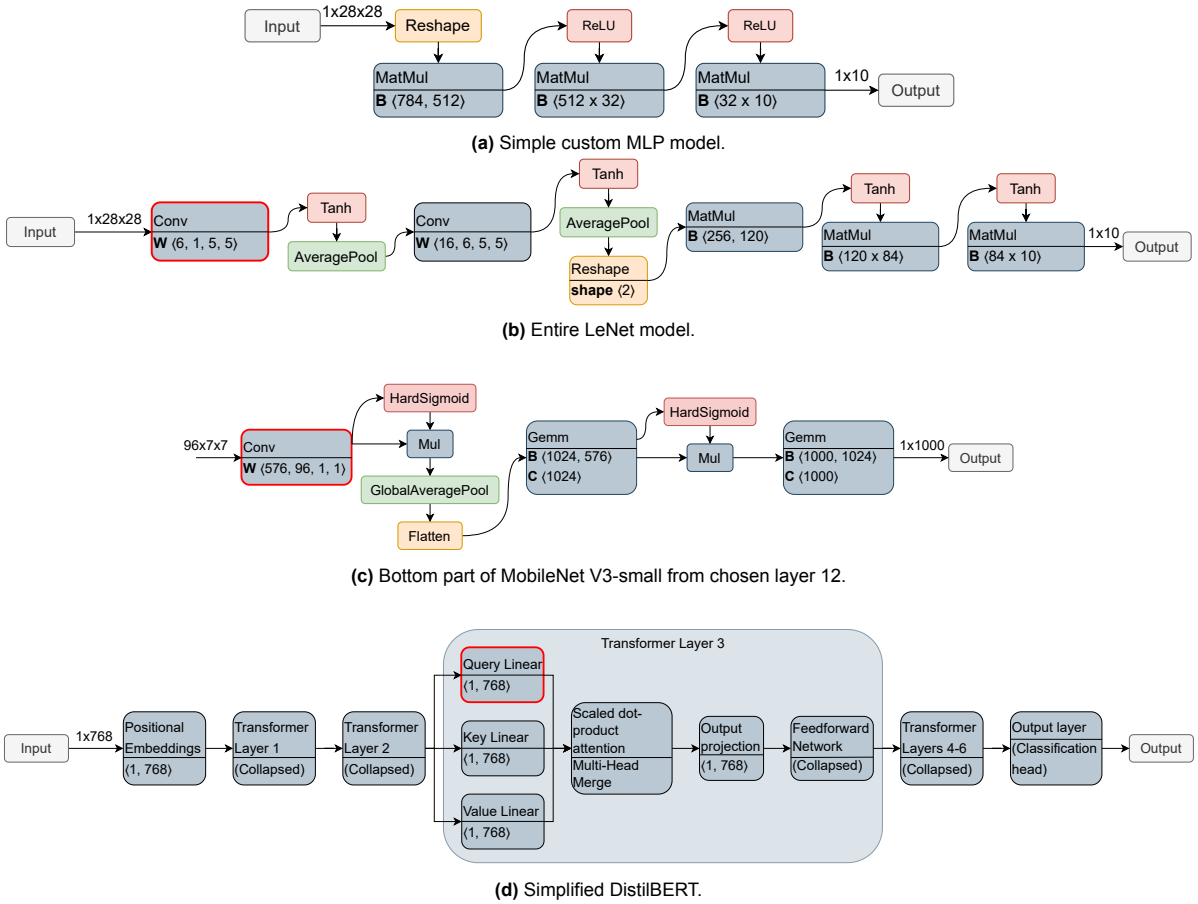
LeNet The second model to be used is LeNet, which is also trained on MNIST but consists of a CNN model. More specifically, the model has a typical MNIST input of 784 values (28×28), following the trend from the first MLP model. The chosen layer was the first of the model, which expects a greyscale image and aims at feature extraction from the raw pixels. Table 3.3 shows an overview of this layer's parameters and their meaning.

Table 3.4: Parameters and their meanings, Layer 12 of MobileNet.

Parameter	Value	Meaning
Output channels	576	Amount of feature maps this layer will produce, expanding from the original number of feature maps
Input channels	96	Amount of feature maps that the layer expects from the previous layer
kernel size	(1, 1)	Height and width of each filter. Called pointwise convolution, it mixes channels but does not process spatial neighbourhood
Input size	(96, 7, 7)	Multiple feature maps of a fixed size
Output size	(576, 7, 7)	An increased depth while maintaining the spatial size due to the kernel size

MobileNet The third workload model to be used was MobileNet, more specifically, the V3-small version of the model. In this model, channels are compressed and expanded for efficiency. Layer 12 of this MobileNet model was chosen to present characteristics different from those of the previously simulated LeNet layer. This layer demonstrates how the $im2cal$ algorithm would work for multichannel inputs, while having a bigger weight matrix size due to the expansion in feature size. An overview of this layer's parameters can be seen in Table 3.4. As can be observed from the kernel size, this layer is part of a bottleneck block, which consists of expansion and projection. Other convolutional layers with a greater kernel size are depthwise separable convolutions that process spatial patterns.

DistilBERT The fourth workload model to be used was DistilBERT. In this model, input sequences are projected into multiple subspaces through separate linear transformations, enabling efficient contextual encoding via self-attention. For scaling purposes, the Query Linear projection of the third Transformer layer was chosen to make a comparable size contrast with the previously simulated MLP using the MNIST dataset. This operation demonstrates how the accelerator handles dense MVM on high-dimensional inputs while providing a weight matrix different from the MNIST example, given the same number of output features as the input. Unlike convolutional layers, this linear transformation has no spatial kernel structure and can be seen as a fully connected layer.



(a) Simple custom MLP model.

(b) Entire LeNet model.

(c) Bottom part of MobileNet V3-small from chosen layer 12.

(d) Simplified DistilBERT.

Figure 3.9: Netron-like structure of neural network models used as workload example in this thesis, with highlighted chosen layers.

Overall, the architecture of the model and layers used can be seen in Figure 3.9. The figure includes the custom simple MLP model, the LeNet model, where the first convolutional layer was simulated, the last few operations of the MobileNet model, starting from the simulated convolutional layer, and lastly, a simplification of the DistilBERT model, highlighting the chosen part of the third layer.

This chapter describes the methodology for simulating CIM architectures. It defines the simulator and details the organisation of the simulated system, including streaming and quantisation techniques. The target architecture implementation is explained, covering signed arithmetic, pipeline design, tiling and utilisation. Tunable parameters for the simulated architecture are outlined, and the chapter concludes with the simulated workloads, including the mapping of CNNs to GEMM operations and the selection of artificial neural network models for simulation.

4

Simulation results

This chapter presents the simulation results of different workloads considering the target tile architecture and the simple system built around it. For verification, a simple MLP model is used to assess the simulator's functionality, considering the baseline parameters presented earlier in Table 3.1. Lastly, other workloads introduced in the previous chapter are simulated with different parameter configurations that exemplify how flexible simulation can be. Results from these simulations are presented and discussed in comparison with metrics such as cycle count, calculated overall tile utilisation, and the number of necessary tiles.

4.1. Functional architectural verification

At each step of simulating the chosen target architecture, DREAM-CIM [16], functional verifications were conducted from a small to a larger scale as architectural building blocks and abstractions were added. Starting from the tile (comprised of the crossbar and its supporting accumulating components), confirming the accumulation logic in place handles tile results correctly, and finally, the storage and communication of (partial) inputs and outputs.

Throughout development, simulation traces provided real-time feedback on correct execution across the defined hardware components, including simulated time and component execution. They allow the verification of each component's computational correctness and task execution order. Even when tasks were scheduled on the same clock cycle in the case of pipelining, the traces made it easy to observe the execution flow across components. Other than that, the simulator engine schedules events in a way that inherently enforces correct hardware execution while recording the simulated time at which specific components perform their tasks. This combination of traces and event scheduling demonstrates the simulator's ability to reproduce correct hardware behaviour and provides a robust basis for verification.

Tiles

The tile marks the simulation platform's core and implements the logic as closely as possible to the target architecture. To verify its correctness, after the input is decoded with all its precision, a simple matrix multiplication algorithm takes place and compares the result with the output computed by the bit-wise logic. For the cycle accuracy, it is enough to verify the number of reported cycles by the simulator, and compare with the expected number of cycles calculated using the tunable parameters and Equation 3.3.

Accumulator

The accumulator component tracks the input offloading to the tiles created during setup. Therefore, to check its correctness, it is also enough to calculate once more the matrix multiplication between the provided input and the weight matrix using a copy of the input read from memory. The correctness of the computation at this state means:

- The tracking of inputs provided to each of the tiles is done correctly;

- The handling of outputs from each tile concerning addition or concatenation to the final resulting array is done correctly.

Memory communication

Two verification methods were applied to ensure the correct memory component simulation. The first is a validity flag for each memory address (holding a byte): A flag indicates the validity of data whenever a specific address is written to. This flag system allowed the verification that enough time had passed between a write and a subsequent read. If the simulator detects a read planned before the first write to that address, it returns an error, indicating that the schedule is unrealistic and incorrect. A run without errors means a component had enough time to write before another component performed a read. This approach has a known flaw: if a component wrote to a memory address more than once, the simulator would not detect cases where "old data" was read. Therefore, another check had to be implemented. Since the memory system was designed simplistically, race conditions could be checked by comparing simulated outputs with equivalent "regular hardware" computations. Pushing the boundaries of these simulated schedules not only detected timing-related issues but also allowed the selection of a standard minimal cycle time for all memory communications, matching the expected memory delay.

From this point, additional memory mechanisms could be incorporated into the framework for a complete and robust implementation. These mechanisms would provide warnings or errors when inconsistencies occur, even in different system configurations. As well as provide a better emulation of a memory component considering a targeted architecture, such as DRAM [67]. However, at this stage of development of the CIM architecture simulation platform, such integration would create boundaries in the system design, shifting the focus from core simulating components to periphery components. Once the simulator reflects a more complete system, integration with memory simulators is the way to add credibility to simulated designs, providing a complete and more accurate simulation of a memory component. This provides cycle-accurate and reliable energy modelling.

4.1.1. Simulating delays

Different components inside the system contribute in different ways to the overall system delay in the number of cycles. Throughout the course of this project, two strategies were assumed for how to simulate such delays: a comprehensive baseline and an optimised system that attempts to reflect a realistic scenario.

Sequential execution

At first glance, the system's delay depended on each of the components finishing their execution before the next component could start, which reflected an extremely inefficient real system. However, the sequential nature of the system was simple to debug, allowing each component to deliver correct outputs before the delay could be improved.

Figure 4.1 shows a UML diagram of how the different system components communicate. The only parallel execution that can be observed here is the tile computation, a core feature attributed to the multiple tile components created by the accumulator. Other component interactions only happen once the component is done with its task.

Table 4.1: Each component and the function dependency of its delay inside the system.

Action	Delay	Assumptions (baseline system)
Writing/Reading input	f(#input bytes)	Input size = 28x28; precision = 4; bandwidth = 10 [$\frac{\text{bytes}}{\text{cycle}}$]: $\lceil \frac{4 \cdot 28 \cdot 28}{8 \cdot 10} \rceil = 40$ cycles.
Start signal	1	Small interrupt delay
Offloading data to tiles	f(#tiles, #tile rows)	Assumed infinite bandwidth: 1 cycle
MAC	Architecture-specific	According to baseline parameters and Equation 3.3: 65 cycles
Receiving data from tiles	f(#tiles, #tile columns)	Assumed infinite bandwidth: 1 cycle
Accumulating tile result	f(#tiles, #tile columns)	One cycle per tile output: 112 cycles
Writing/Reading output	f(#output bytes)	Output size = 512; output precision = 16; bandwidth = 10 [$\frac{\text{bytes}}{\text{cycle}}$]: $\lceil \frac{16 \cdot 512}{8 \cdot 10} \rceil = 103$ cycles

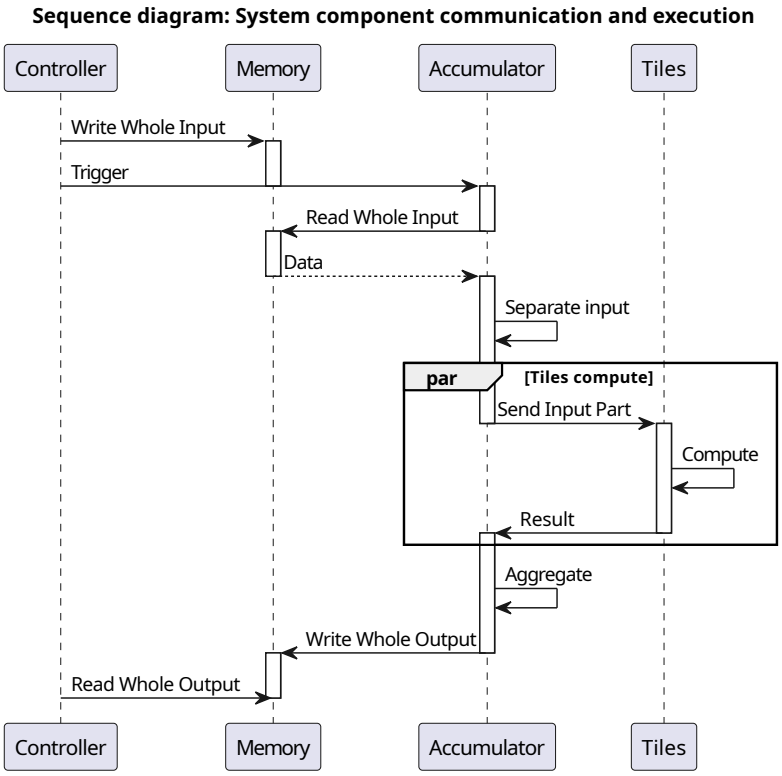


Figure 4.1: UML sequence diagram of the sequential system execution considering the functionality of each component.

Table 4.1 summarises the system delay dependencies, their assumptions and calculated values. It can be seen that apart from parts with high delay, this communication behaviour includes non-idealities. The first non-ideality is the simultaneous input communication from the accumulation units to all tiles, which is unrealistic when many tiles are involved. The second is the simultaneous result written back to the accumulator; since the tile delay is fixed, all results would also return at the same time. The assumptions for the system can then be summarised as:

- Infinite bandwidth when offloading work to the tiles, where, independent of how many tiles the system has, their activation and deactivation would happen instantly;
- Data movement and computation happen sequentially, where each memory access or computation completes before the next begins.

Considering the target architecture baseline parameters and the first layer of the simple MLP model with the MNIST 28×28 greyscale input, the system would take a total of 465 cycles to finish execution. Loading data simultaneously into all tiles would be highly efficient; however, this assumes an unrealistic infinite bandwidth. Moreover, if computation could overlap with data transfers, reducing component idle time, significant cycle savings could be achieved, since most of the total execution time is spent on reading and writing data.

Pipelining

Because of previously shown drawbacks and non-idealities, a modified system now considers:

- More realistic tile activation and writing back: This communication model is necessary to simulate a real scenario involving the number of tiles to write data to, impacting the system's performance due to communication delays;
- Pipeline execution with communication delays: Using an input and output buffer instead of a singular memory component. As a result, less memory is required to run the system since only part of the data is saved in the buffer at a specific time;
- Due to a realistic tile activation, results can also be accumulated while other tiles finish computing.

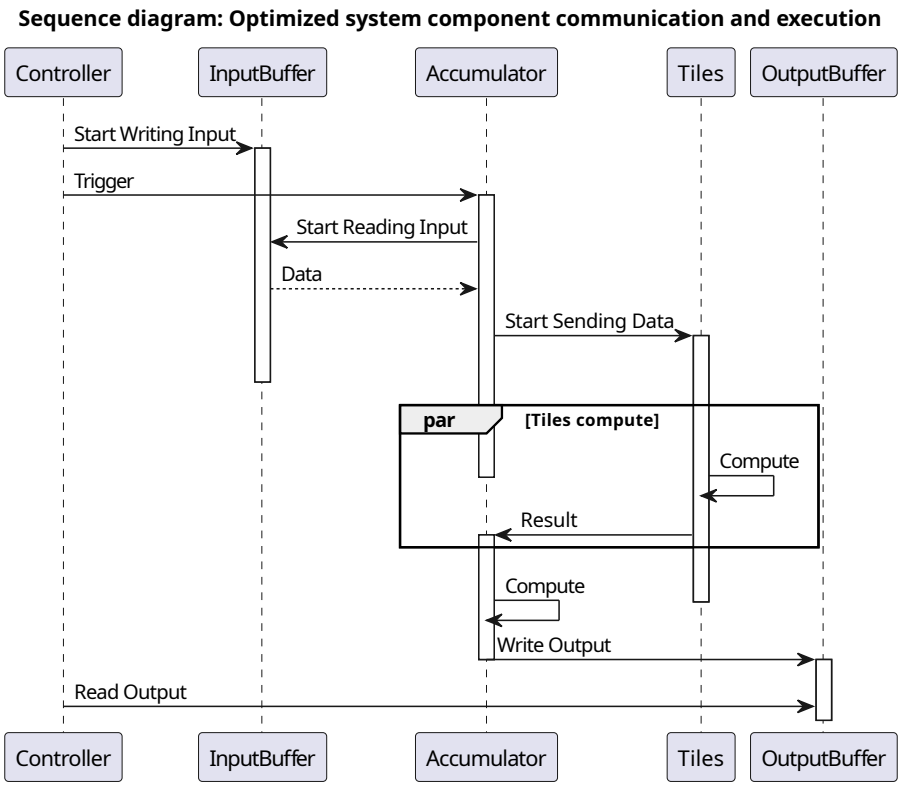


Figure 4.2: UML sequence diagram of the target system considering minimal delays due to pipelining and buffering.

With a more efficient system, a delay calculation ensured efficient use of the architecture, also considering the tunable parameters that designers can use. The sequence diagram in Figure 4.2 gives an overview of the different system components communicating together and the target architecture inside a system being simulated as efficiently as possible. Unlike the original, this system has delays as if the memory were split into input and output buffers. At the same time, a first-in-first-out (FIFO) queue would allow data to be streamed inside the components, and these could start computing without having to wait for the whole input to be written.

The pipeline design changes the assumptions for each of the actions that happen inside the system, considering all its components. For this reason, Table 4.2 shows the changed assumptions taken in the system for each core considered action.

Table 4.2: Modified assumptions for each component in an optimised system.

Action	Changed Assumptions (optimised system)
Writing/Reading input	Data transfer delays are hidden in tile computation time, extra margin: 2 cycles
Start signal	Same small interrupt delay
Offloading data to tiles	Finite bandwidth, one cycle per tile offload: 112 cycles
MAC	The same, given parameters and Equation 3.3: 65 cycles
Receiving data from tiles	Follows from tile offloading offset, extra margin: 2 cycles
Accumulating tile result	Aggregation done within tile offloading offset, extra margin: 2 cycles
Writing output	Output is buffered out of the system, extra margin: 2 cycles
Reading output	Considering time to last output, same reading delay: 103 cycles

In this optimised system, communication latencies are largely hidden by concurrent computation over pipeline stages. Therefore, a fixed two-cycle synchronisation overhead is systematically assumed, accounting for setup and buffering actions.

Finally, the comparison between this optimised and the initial assumed system can be seen in Figure 4.3.

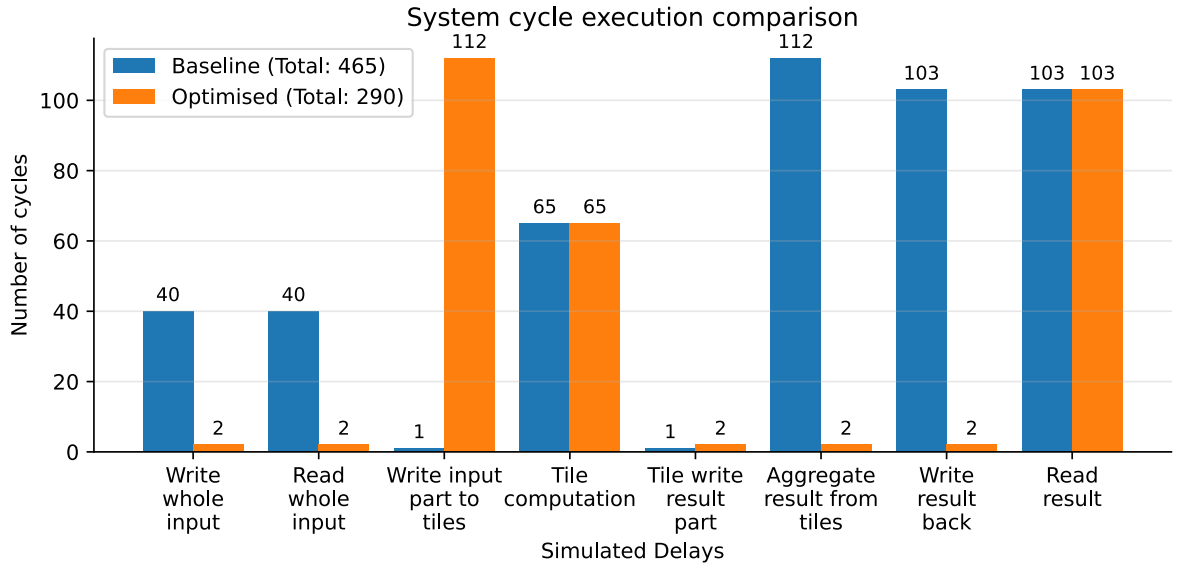


Figure 4.3: Broken down cycle count of baseline and optimised simulated accumulation.

The figure shows a drop in total execution time to ∼60% of the baseline system, thanks to data buffering and pipelined computations. Writing the input to tiles with an offset, considering finite bandwidth, hides the aggregation delay without impacting the total computation time. The most significant delay reduction comes from buffering, which improves data reads and writes throughout the system. The output read time remains unchanged, since the delay is measured until the last result is read.

In the simulator, a request can be sent to each tile as a scheduled event. Therefore, to map a realistic workload and take care of communication costs, if multiple tiles are necessary for a computation, each is written to in a different cycle. This offset allows for a small level of contention at the accumulator level. This concurrent behaviour reflects real hardware scenarios, where not all computing units are done simultaneously, and data transfers have a realistic, limited bandwidth. As it is possible to see, because of the concurrent execution, data movement delays are minimal, while maintaining a realistic execution, the compound computation time of all tiles was found to follow the relation:

$$t_{\text{tile}} = N \cdot D + C \tag{4.1}$$

Where t_{tile} is the total time to complete all tile computations assuming pipelined input, N is the number of tiles, D is the input delay for data to reach the tile, and C is the computation time per tile. The impact of this calculated delay in the overall schedule of the workload can also be seen in Listing 4.1, where `accumulator_delay` includes the compound tile computation delay, accumulation and the start of output write-back to memory.

Listing 4.1: Controller Schedule of multi-patch workload.

```
for (size_t patch_id = 0; patch_id < workload.total_patches; patch_id++) {

    // Schedule patch write to memory, always reusing the input address
    uint64_t patch_addr = config.addr_input;
    ctr.ScheduleEvent(current_time, [&ctr, patch_addr, patch_bytes]() {
      ctr.WriteToMemory(patch_addr, patch_bytes);
    });

    // Schedule the start of the computation after the input is written into memory
    ctr.ScheduleEvent(current_time + write_input_delay, [&ctr, &config]() {
      std::vector<uint8_t> go_signal{};
      ctr.WriteToAccumulator(config.addr_signal, go_signal);
    });

    // Schedule to read the output after the accumulation happened
```

```
    ctr.ScheduleEvent(
        current_time + write_input_delay + accumulator_delay, [&ctr, &config, &workload]() {
        ctr.ReadFromMemory(config.addr_output, workload.o_size);
    });

    current_time += patch_interval;
```

4.2. Workload Evaluation

Different topologies can be explored now that the simulated architecture is in place. Adding degrees of freedom for designers to experiment with different workloads for various system variables.

The simulated framework reads the provided weight matrix/convolution kernel, performs the flattening when needed and determines the amount of DREAM-CIM [16] tiles necessary. Then, the `im2col` algorithm is applied for different inputs when required, to generate an MVM problem. Therefore, for different workloads, it is intuitive to analyse the necessary hardware and execution cycles for different tunable parameters. Following the performed cycle delay analysis discussed in section 4.1.1, comparing workloads will consider the optimised pipelined system. This choice was made to bring credibility to the results, as it compares with the scale of real-world execution times. Furthermore, the comparison aims to analyse the behaviour of the tiles following the target architecture and its communication, rather than the full-system simulation.

Given the imbalances in simulated against real constraints, such as digital circuit timing, area and energy consumption, which have not yet been implemented into the simulator, a complete DSE was not conducted. Instead, a few configurations were chosen to represent different system configurations. All the configurations are then compared to the baseline tile configuration shown in Table 3.1.

Utilisation

The utilisation of a tiled matrix multiplication presented in Equation 3.4 can be further specified so utilisation can be used as a metric for the different analysed workloads. Although it is fair to assume that the number of crossbar columns is a multiple of the weight precision, to be thorough, the equation that defines the utilisation in a single tile can be expressed as:

$$\mathsf{U}_{\text{crossbar}} = \frac{m \cdot n}{R \cdot \left\lfloor \frac{C}{P} \right\rfloor} \tag{4.2}$$

Where $m \cdot n$ is the size of the weight matrix assigned to the specific crossbar, R is the number of crossbar rows, C is the number of crossbar columns, and P is the precision of the weights. Then, for the overall utilisation across all crossbars:

$$\mathsf{U} = \frac{M \cdot N}{N_{\text{crossbars}} \cdot R \cdot \left\lfloor \frac{C}{P} \right\rfloor}, \text{ with: } N_{\text{crossbars}} = \left\lceil \frac{M}{R} \right\rceil \cdot \left\lceil \frac{N}{\left\lfloor \frac{C}{P} \right\rfloor} \right\rceil \tag{4.3}$$

M·N is now the size of the whole weight matrix.

4.2.1. Simple MLP workload

The first workload to be simulated was a simple MLP workload. More specifically, the first layer of a simple MLP model that uses the MNIST dataset. A whole inference of this simple MLP workload was also simulated throughout the thesis. However, only the first layer is considered for comparison with other workloads. The chosen layer has 784 input features and 512 output features, corresponding to the entry point, with an input of a 28×28 greyscale handwritten “2”.

Given the chosen parameter configurations, Table 4.3 show the resulting simulated metrics. It is worth mentioning that the simple MLP workload with baseline parameters was used as the example to demonstrate system delays in section 4.1.

From the table, it is possible to see how, for smaller arrays, the utilisation is higher, showing that for an array with $\frac{1}{4}$ of the size, the number of cycles is less than three times higher. Another observation is

Table 4.3: Resulting metrics for simple MLP workload.

FC1(MLP)	Baseline	Small Arrays	Increased Precision	Balanced
#cycles	189	493	717	253
#tiles	112	416	448	112
Utilisation(%)	87.5	94.2	87.5	87.5

that for the chosen balanced configuration, a slight increase of $\frac{1}{3}$ of the cycles allows the architecture to handle double the precision for both inputs and weights, which confirms the choice of a balanced configuration and sets the trend of changes for this workload to increase the number of array columns, especially when considering higher precisions.

4.2.2. LeNet

When executing the first convolutional layer of LeNet, it was possible to observe that only one tile is necessary to fit all the synaptic weights. After flattening, the weight matrix is of shape 25×6 values, which comfortably fits in the baseline target architecture 128×128-bit crossbar for bit precisions up to 16-bit without any recoding or different writing mechanisms.

Table 4.4: Resulting metrics for LeNet CNN workload.

Conv1(LeNet)	Baseline	Small Arrays	Increased Precision	Balanced
#cycles	44,928	44,928	155,520	81,792
#tiles	1	1	1	1
Utilisation(%)	3.66	14.65	14.65	3.66

Table 4.4 summarises the resulting metrics for the first convolutional layer of the LeNet workload. The very low utilisation, with its lowest at 3.66%, is attributed to the flattened weight matrix occupying such a small space, leading to the number of cycles skyrocketing. Although the utilisation increases for smaller arrays or increased precision, it is still far from being helpful and efficient to use such a workload with the considered hardware without extra programmability of the array.

Such programmability can be, for example, the copy of the 25×6 matrix multiple times in the same array. Three weight matrices would fit in a single array for the baseline size, even if only row-wise duplication were considered.

Without added programmability changes, the design trend observed is to decrease the size of the arrays even further to increase utilisation while reducing the area. However, in a realistic full-fledged accelerator design, this model would only be used with some extra programmability in the accelerator architecture, allowing data duplication within the same tile to increase parallelism and utilisation. Therefore, this workload shows how the flexibility of mapping the data differently to crossbars, even if minimal, should exist and be explored in an accelerator design.

4.2.3. MobileNet

The MobileNet model is usually trained on the ImageNet dataset, which differs from MNIST. Unlike MNIST, which contains simple greyscale handwritten digits, ImageNet includes images with more complex features, diverse objects, and multiple colour channels. These differences mean the MobileNet model is designed to process more input channels from the beginning, allowing it to capture richer information. In addition, MobileNet is inherently a larger and more sophisticated model than those commonly used for simpler image classification tasks like MNIST, making it well-suited for handling complex, real-world images.

Table 4.5: Resulting metrics for MobileNet CNN workload.

Conv12(MobileNet)	Baseline	Small Array	Increased Precision	Balanced
#cycles	4,655	7,301	16,709	7,791
#tiles	18	72	72	18
Utilisation(%)	75	75	75	75

The chosen convolutional layer of MobileNet is used to increase the depth of the feature maps. As a result, the weight matrix written to the tiles in the architecture has a shape of 96×576 after "flattening" (due to the kernel size of (1,1), this operation is just a reshape). Due to its greater width, the number of output features is higher than in other workloads. For the baseline architecture, 18 tiles are used, where only 96 rows of the said tiles are filled. This matrix size leads to a relatively low, but acceptable, utilisation of 75%. Table 4.5 shows this workload's utilisation and other metrics for different parameters.

From the table, it is possible to see how the utilisation remains constant for the different configurations. This constancy is due to the topology of the weight matrix, which has only 96 rows when flattened. When using a 128-row crossbar array, only 96 rows are occupied. In contrast, when using a 64-row crossbar, another set of tiles is necessary to split the weight matrix, where one has 100% utilisation and the other has 50%, averaging 75% ($\frac{64+32}{64+64}$), with all columns occupied.

Because the utilisation is the same for all tested workloads, the trend considering only this workload would be towards an intermediate size, smaller than the baseline and bigger than the small array configuration. That is done while maintaining double the number of columns in the crossbar array to support greater precision without impacting performance.

4.2.4. DistilBERT

Table 4.6: Resulting metrics for DistilBERT workload.

Query Linear 3 (DistilBERT)	Baseline	Small Array	Increased Precision	Balanced
#cycles	221	653	845	285
#tiles	144	576	576	144
Utilisation(%)	100	100	100	100

To demonstrate the architecture's performance for a different workload, layer three of the DistilBERT NLP model was used, more specifically, a Query Linear operation. The weight matrix is of shape 768×768, which means the number of output features will be the same as the number of inputs. Table 4.6 shows the metrics for different parameter configurations.

From the table, it is possible to see the perfect utilisation for all given configurations, this is attributed to the input being a multiple of 32, occupying all available space and making the most out of the accumulating hardware.

Because the increased precision and the small array configuration lead to an exploding number of tiles and cycles, the trend lies in increasing the number of columns compared to the number of rows. An increased number of columns is already the case for the balanced configuration, which allows for a slight increase in the number of cycles for double the precision compared to the baseline configuration.

4.2.5. Comparison

Following the presented results for each of the workloads considering the different configurations, a comparison including the different workloads together gives an overall view of the assumed architectures, and ultimately further verification of the simulating platform, where the Pearson Correlation Coefficient (r) is used to assess the correlation between each metric pair.

The goal of this comparison is to exemplify that the simulating platform has versatile configuration parameters and is not meant to be a complete DSE. For simplicity and exemplification of the simulated system, such metrics included in the simulation are enough to direct designers towards a more efficient and general architecture of neural network accelerators.

Three different associations were made considering the calculated and observed metrics. The first is a relation between the number of cycles and the number of tiles, since these are closely related, being the number of tiles one of the significant contributors to the total cycle count of the system, together with the tile execution delay. The second is the number of cycles together with the utilisation, highlighting how a high utilisation needs to be the standard for an efficient system. Lastly, the utilisation against the number of tiles, analysing how the spatial distribution influences performance.

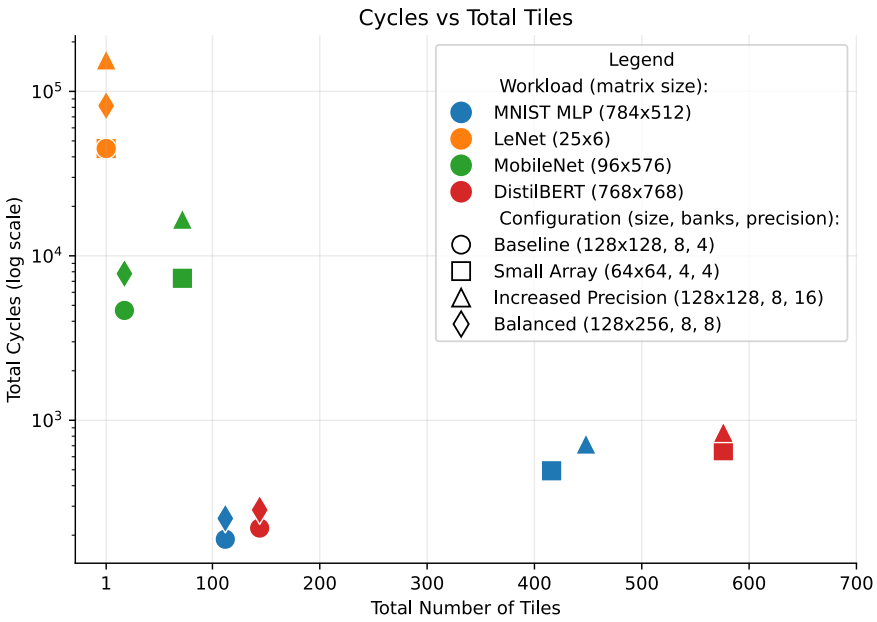


Figure 4.4: Relation of cycle count against number of tiles for each workload and parameter configuration.

Number of Cycles vs. Number of Tiles

Figure 4.4 shows the relation between the number of cycles and the number of tiles. The graph uses a logarithmic scale for the total number of cycles to facilitate visualisation, which can be interpreted as an even higher exponential trend of the scattered points. The figure shows that the baseline configuration is the most optimal for all of the workloads, presenting a good trade-off between the total number of cycles and the number of tiles. Furthermore, the same number of tiles was observed between pairs of configurations, apart from the MNIST MLP between "small array" and "increased precision" configurations. This difference is due to a better utilisation of the tiles (87.5% vs. 94.2%), leading to a 32 tile difference between them. LeNet does not benefit from more tiles, because the number of weights for this layer is small and fits in all pre-defined parameter configurations.

The expected result was that these two metrics were negatively correlated. As the number of tiles increases, the total number of cycles should decrease. The calculated correlation coefficient is $r = -0.44$, which indicates a moderate negative correlation. The negative correlation is as expected; however, the fact that it is a weak correlation is attributed to the accumulator to tile communication. This proves that dividing the input and sending it to each of the tiles using the presented system topology impacts performance and should be considered in the simulation.

The system is assumed to take a cycle to call and send data to each of the tiles. Because of such a simplistic assumption, the communication impact is greater for smaller tiles (represented by the figures' squares), which also have a smaller delay. Apart from that, the number of cycles assigned for memory operations and activations was observed not to have a noticeable impact on the correlation between the metrics. For this reason, through the rest of the comparison, they are regarded as a simple offset.

Number of Cycles vs. Utilisation

Same as the last relation, Figure 4.5 also has a logarithmic scale representation of the number of cycles. The data points for each workload are almost grouped due to the differences in size and type of workload, but it is clear how the trend holds between workloads and parameter configurations. Once more, the most efficient configuration was proven to be baseline, with the least number of cycles and higher utilisation. This is just not the case for LeNet, because of how small the layer is, even with higher utilisation, the number of cycles is the same.

It was expected that the correlation between the number of cycles and the utilisation would be negative, since fuller tiles would mean more data would be computed together without wasting computing power

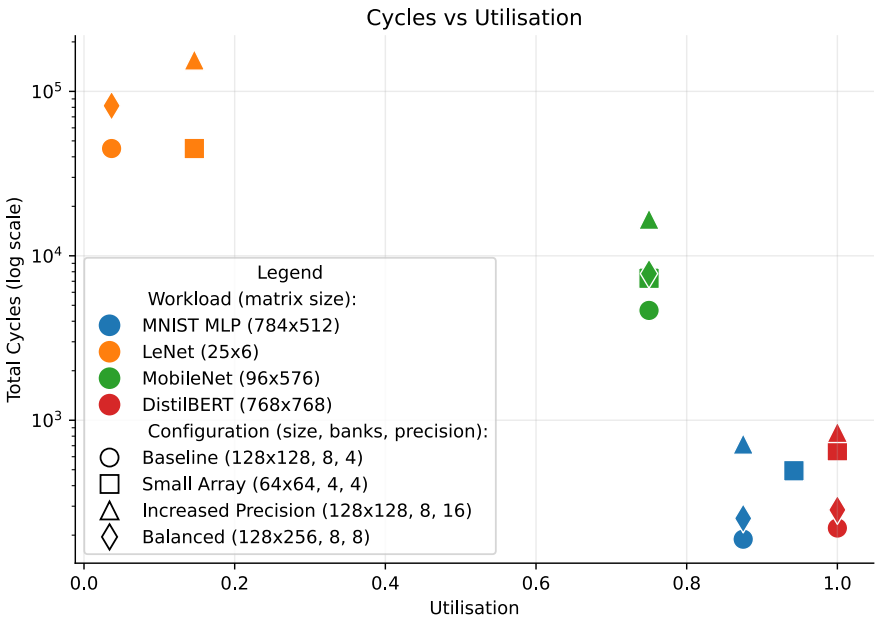


Figure 4.5: Relation of utilisation against cycle count for each workload and parameter configuration.

on padding data, lowering the total cycle count. As expected, the correlation coefficient between the number of cycles and the utilisation is $r = -0.8$, a strong negative correlation.

Utilisation vs. Number of Tiles

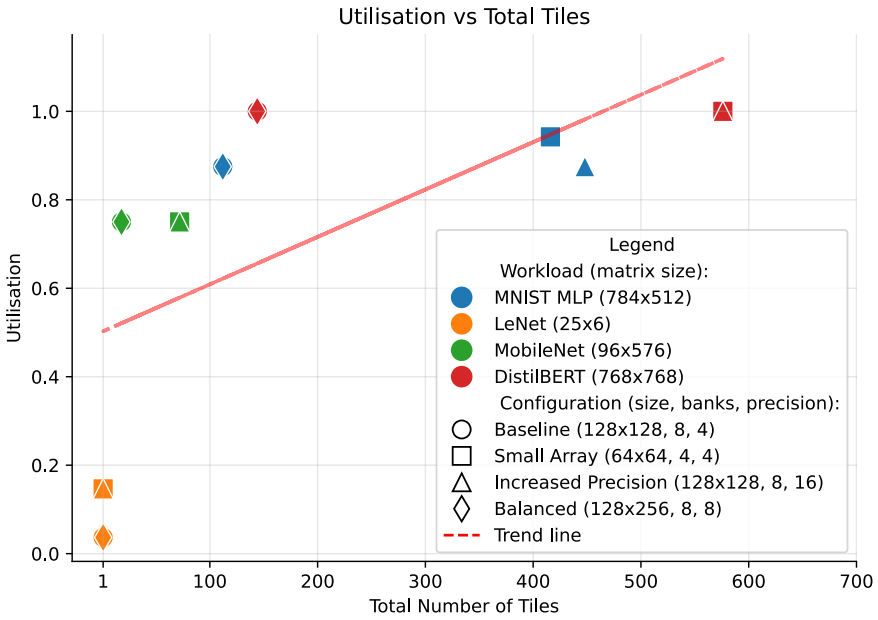


Figure 4.6: Relation of utilisation against number of tiles for each workload and parameter configuration.

The last calculated correlation is between the utilisation and the number of tiles needed. The plot exemplifying this relation, together with a trend line, can be seen in Figure 4.6. As expected, the correlation between the two metrics is positive and has a value of $r = 0.62$, which is moderate. The correlation is associated with how bigger weight matrices tend to fill the crossbars inside the tiles more efficiently, leading to less edge waste considering the tiling principle used. In the figure, it is possible to see how all baseline and balanced configurations overlap. This occurs because between the two configurations,

both the number of columns and the precision were doubled, leading to the same utilisation as long as the array size is a multiple of the precision.

4.3. Simulator Performance

Table 4.7: Hardware/Software used to run the framework and all tests.

Component	Specification
CPU	Intel Core i7-13700H, 14 cores, 2.4 GHz
RAM	15.6 GB (WSL2 8GB) 5200 MT/s
Storage	477 GB NVMe SSD
Operating System	Host: Windows 11 + WSL2: Ubuntu 24.04 LTS
Compiler	GCC 13.3.0

Considering the CIM architecture simulation platform developed by the Computer Engineering group, running the tests for all configurations and workloads together took 13 seconds, considering the hardware/software summarised in Table 4.7. The execution time includes time to load a weight matrix from a CSV file for each of the 16 tests and distribute values to the tiles, which accounts for what was named: setup time, before the event engine starts its execution.

This chapter presents the simulation results of the CIM architecture. It began with a functional verification of the tile and defined system architecture, including delay handling. Then it evaluates performance across various workloads, considering a set of tunable parameters. The results include comparisons between workloads and report the simulator's overall performance.

5

Conclusion

This thesis set out to develop an event-based computing-in-memory (CIM) architecture framework simulator, capable of streamlining design-space exploration (DSE) to address the demand for accelerator systems in the current AI landscape. The main contributions of this work are: (i) the definition of system requirements for an accelerator system considering a target microarchitecture, (ii) the implementation of an event-based CIM architecture simulator platform, and (iii) demonstrating the platform's support for multiple types of workload and flexibility through tunable parameters.

The simulation framework was demonstrated through a target architecture that uses a digital 8T-SRAM computing memory array integrated into an accelerator system. The framework brings high-level performance estimation with near circuit-level analysis by providing cycle-dependent modelling, configurable architectural parameters, and flexibility to support different workloads.

The development of the framework involved several stages: (1) the bit-wise logic of the accelerator microarchitecture was implemented and simulated, (2) an accelerator system encompassing control, storage, and additional computing components was instantiated and evaluated, (3) neural network workloads were subsequently chosen to perform an area, delay, and utilisation DSE at the high-level accelerator design. Through this process, the framework proved capable of modular and scalable modelling of CIM systems, supporting workload-specific analysis. The overlap of high-level architectural exploration with low-level physical constraints reveals bottlenecks and trade-offs in precision, array size, and buffering.

The methodology and results presented in this thesis highlight the relevance of CIM and the potential of event-based simulation to guide accelerator design, reduce development costs, and inform hardware decisions. The simulator offers a straightforward approach for exploring digital and analog CIM design implementations, capturing performance metrics such as throughput, latency and utilisation. The framework's flexibility allows for experiments with emerging non-volatile memory technologies in different architectural configurations and system topologies, offering a solid basis for accelerator designs using analog CIM integrated in a digital design.

Experimental simulation results between a pipelined and a non-pipelined system showed that pipelined designs achieve much lower total execution time (with a reduction of $\sim$40% compared to the baseline design), highlighting the importance of incorporating data streaming hardware into accelerator designs. Furthermore, analysis of LeNet CNN layers highlighted the need for programmability in accelerators (e.g. by duplicating kernel data across multiple arrays and replicating inputs to improve utilisation and reduce computation time). Finally, results from other workloads showed that rectangular crossbars (rather than square) led to more efficient systems, as most workloads require precision greater than the bit capacity of a single memory cell.

In conclusion, this thesis presents an event-based simulation framework as a robust and versatile tool for CIM research. Combining detailed component modelling, workload adaptability, and performance metrics output enables DSE that informs hardware decisions while reducing experimentation costs and

development time, paving the way for more efficient next-generation accelerator systems.

5.1. Future prospects

Future work should extend this framework in the following directions: (1) incorporate realistic analog memory behaviour, (2) implement energy consumption metrics, and (3) add support for asynchronous workloads (such as SNNs). Such changes would improve the relevance of the simulator framework by aligning with emerging neuromorphic computing trends. Lastly, improving the simulator modularity, separating workload and hardware description, would allow for easier integration with existing simulators and create a clear separation of concerns.

Together, the listed extensions would make the framework a comprehensive and full-fledged platform for CIM research. The simulator's modularity, accuracy, and flexibility make it a practical tool for investigating the next generation of CIM architectures, bridging the gap between idea and implementation.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton. “Deep learning”. en. In: *Nature* 521.7553 (May 2015). Publisher: Nature Publishing Group, pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539.

[2] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou. “Memory devices and applications for in-memory computing”. en. In: *Nature Nanotechnology* 15.7 (July 2020). Publisher: Nature Publishing Group, pp. 529–544. ISSN: 1748-3395. DOI: 10.1038/s41565-020-0655-z.

[3] W. A. Wulf and S. A. McKee. “Hitting the memory wall: implications of the obvious”. en. In: *ACM SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588.

[4] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie. “Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey”. In: *Proceedings of the IEEE* 108.4 (Apr. 2020), pp. 485–532. ISSN: 1558-2256. DOI: 10.1109/JPROC.2020.2976475.

[5] D. Ielmini. “Resistive switching memories based on metal oxides: mechanisms, reliability and scaling”. en. In: *Semiconductor Science and Technology* 31.6 (May 2016). Publisher: IOP Publishing, p. 063002. ISSN: 0268-1242. DOI: 10.1088/0268-1242/31/6/063002.

[6] M. Kang, S. K. Gonugondla, A. Patil, and N. R. Shanbhag. “A Multi-Functional In-Memory Inference Processor Using a Standard 6T SRAM Array”. In: *IEEE Journal of Solid-State Circuits* 53.2 (Feb. 2018), pp. 642–655. ISSN: 1558-173X. DOI: 10.1109/JSSC.2017.2782087.

[7] N. Verma and A. P. Chandrakasan. “A 256 kb 65 nm 8T Subthreshold SRAM Employing Sense-Amplifier Redundancy”. In: *IEEE Journal of Solid-State Circuits* 43.1 (Jan. 2008), pp. 141–149. ISSN: 1558-173X. DOI: 10.1109/JSSC.2007.908005.

[8] T. M. Taha, R. Hasan, C. Yakopcic, and M. R. McLean. “Exploring the design space of specialized multicore neural processors”. In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*. ISSN: 2161-4407. Aug. 2013, pp. 1–8. DOI: 10.1109/IJCNN.2013.6707074.

[9] N. Bruschi, G. Haugou, G. Tagliavini, F. Conti, L. Benini, and D. Rossi. “GVSoC: A Highly Configurable, Fast and Accurate Full-Platform Simulator for RISC-V based IoT Processors”. In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. arXiv:2201.08166 [cs, eess]. Oct. 2021, pp. 409–416. DOI: 10.1109/ICCD53106.2021.00071.

[10] H. Ishiwara. “Ferroelectric Random Access Memories”. In: *Journal of Nanoscience and Nanotechnology* 12.10 (Oct. 2012), pp. 7619–7627. DOI: 10.1166/jnn.2012.6651.

[11] A. D. Kent and D. C. Worledge. “A new spin on magnetic memories”. en. In: *Nature Nanotechnology* 10.3 (Mar. 2015). Publisher: Nature Publishing Group, pp. 187–191. ISSN: 1748-3395. DOI: 10.1038/nnano.2015.24.

[12] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. “Phase-change random access memory: A scalable technology”. en. In: *IBM Journal of Research and Development* 52.4.5 (July 2008), pp. 465–479. ISSN: 0018-8646, 0018-8646. DOI: 10.1147/rd.524.0465.

[13] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. “ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. ISSN: 1063-6897. June 2016, pp. 14–26. DOI: 10.1109/ISCA.2016.12.

[14] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, and D. S. Milojicic. “PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 715–731. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304049.

[15] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. “PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory”. In: *SIGARCH Comput. Archit. News* 44.3 (June 2016), pp. 27–39. ISSN: 0163-5964. DOI: 10.1145/3007787.3001140.

[16] A. el Arrassi, L. Huijbregts, M. D. Gomony, A. Gebregiorgis, F. Catthoor, M. Taouil, R. Joshi, and S. Hamdioui. “DREAM-CIM: A Digital SRAM-Based CIM Accelerator for Energy- and Area-Efficient Edge AI”. In: *IEEE Transactions on Circuits and Systems for Artificial Intelligence* (2025), pp. 1–11. ISSN: 2996-6647. DOI: 10.1109/TCASAI.2025.3579709.

[17] D. Ielmini and H.-S. P. Wong. “In-memory computing with resistive switching devices”. en. In: *Nature Electronics* 1.6 (June 2018). Publisher: Nature Publishing Group, pp. 333–343. ISSN: 2520-1131. DOI: 10.1038/s41928-018-0092-2.

[18] Y. Kim, Y. Zhang, and P. Li. “A digital neuromorphic VLSI architecture with memristor crossbar synaptic array for machine learning”. In: *2012 IEEE International SOC Conference*. ISSN: 2164-1706. Sept. 2012, pp. 328–333. DOI: 10.1109/SOCC.2012.6398336.

[19] *Circuit Simulation*. en. URL: https://www.cadence.com/en_US/home/tools/custom-ic-analog-rf-design/circuit-simulation.html (visited on 06/23/2025).

[20] F. Cüppers, S. Menzel, C. Bengel, A. Hardtdegen, M. von Witzleben, U. Böttger, R. Waser, and S. Hoffmann-Eifert. “Exploiting the switching dynamics of HfO2-based ReRAM devices for reliable analog memristive behavior”. In: *APL Materials* 7.9 (Sept. 2019), p. 091105. ISSN: 2166-532X. DOI: 10.1063/1.5108654.

[21] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst. “ZigZag: Enlarging Joint Architecture-Mapping Design Space Exploration for DNN Accelerators”. In: *IEEE Transactions on Computers* 70.8 (Aug. 2021), pp. 1160–1174. ISSN: 1557-9956. DOI: 10.1109/TC.2021.3059962.

[22] T. Andrulis, J. S. Emer, and V. Sze. “CiMLoop: A Flexible, Accurate, and Fast Compute-In-Memory Modeling Tool”. In: *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. arXiv:2405.07259 [cs]. May 2024, pp. 10–23. DOI: 10.1109/ISPASS61541.2024.00012.

[23] A. Lu, X. Peng, W. Li, H. Jiang, and S. Yu. “NeuroSim Simulator for Compute-in-Memory Hardware Accelerator: Validation and Benchmark”. English. In: *Frontiers in Artificial Intelligence* 4 (June 2021). Publisher: Frontiers. ISSN: 2624-8212. DOI: 10.3389/frai.2021.659060.

[24] R. Pelke, J. Cubero-Cascante, N. Bosbach, N. Degener, F. Idrizi, L. M. Reimann, J. M. Joseph, and R. Leupers. *Optimizing Binary and Ternary Neural Network Inference on RRAM Crossbars using CIM-Explorer*. arXiv:2505.14303 [cs]. May 2025. DOI: 10.48550/arXiv.2505.14303.

[25] J. Lowe-Power et al. *The gem5 Simulator: Version 20.0+*. arXiv:2007.03152 [cs]. Sept. 2020. DOI: 10.48550/arXiv.2007.03152.

[26] M. Zahedi, M. A. Lebdeh, C. Bengel, D. Wouters, S. Menzel, M. Le Gallo, A. Sebastian, S. Wong, and S. Hamdioui. “MNEMOSENE: Tile Architecture and Simulator for Memristor-based Computation-in-memory”. In: *J. Emerg. Technol. Comput. Syst.* 18.3 (Jan. 2022), 44:1–44:24. ISSN: 1550-4832. DOI: 10.1145/3485824.

[27] J. von Neumann. “First draft of a report on the EDVAC”. In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75. ISSN: 1934-1547. DOI: 10.1109/85.238389.

[28] G. E. Moore. “Cramming more components onto integrated circuits”. en. In: *Electronics* 38.8 (1965).

[29] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. ISSN: 1558-173X. DOI: 10.1109/JSSC.1974.1050511.

[30] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. “Dark silicon and the end of multicore scaling”. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. ISSN: 1063-6897. June 2011, pp. 365–376.

[31] W. S. McCulloch and W. Pitts. “A logical calculus of the ideas immanent in nervous activity”. en. In: *Bulletin of Mathematial Biophysics* 5 (1943), pp. 115–133.

[32] F. Rosenblatt. “The perceptron: A probabilistic model for information storage and organization in the brain.” en. In: *Psychological Review* 65.6 (1958). Publisher: American Psychological Association (APA), pp. 386–408. ISSN: 1939-1471, 0033-295X. DOI: 10.1037/h0042519.

[33] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning representations by back-propagating errors”. en. In: *Nature* 323.6088 (Oct. 1986). Publisher: Nature Publishing Group, pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0.

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “ImageNet classification with deep convolutional neural networks”. In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386.

[36] J. L. Elman. “Finding Structure in Time”. en. In: *Cognitive Science* 14 (1990), pp. 179–211.

[37] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. *Attention Is All You Need*. arXiv:1706.03762 [cs]. Aug. 2023. DOI: 10.48550/arXiv.1706.03762.

[38] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. “Binarized neural networks”. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS’16. Red Hook, NY, USA: Curran Associates Inc., Dec. 2016, pp. 4114–4122. ISBN: 978-1-5108-3881-9.

[39] W. Maass. “Networks of spiking neurons: The third generation of neural network models”. en. In: *Neural Networks* 10.9 (Dec. 1997). Publisher: Elsevier BV, pp. 1659–1671. ISSN: 0893-6080. DOI: 10.1016/s0893-6080(97)00011-7.

[40] S. Han, H. Mao, and W. J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. arXiv:1510.00149 [cs]. Feb. 2016. DOI: 10.48550/arXiv.1510.00149.

[41] A. Davis, M. Schuette, D. Franklin, A. Thomasian, N. Mekhiel, Y. Hu, R. J. Baker, and J. Smith. “In Praise of Memory Systems: Cache, DRAM, Disk”. en. In: *Memory Systems*. Elsevier, 2008, pp. i–ii. ISBN: 978-0-12-379751-3. DOI: 10.1016/B978-0-12-379751-3.50036-9.

[42] M. L. Bushnell. “DSP-Based Analog and Mixed-Signal Test”. en. In: *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Ed. by M. L. Bushnell and V. D. Agrawal. Boston, MA: Springer US, 2002, pp. 309–384. ISBN: 978-0-306-47040-0. DOI: 10.1007/0-306-47040-3_10.

[43] T. Mikolajick, M. Salinga, M. Kund, and T. Kever. “Nonvolatile Memory Concepts Based on Resistive Switching in Inorganic Materials”. en. In: *Advanced Engineering Materials* 11.4 (2009). _eprint: https://advanced.onlinelibrary.wiley.com/doi/pdf/10.1002/adem.200800294, pp. 235–240. ISSN: 1527-2648. DOI: 10.1002/adem.200800294.

[44] A. V. Khvalkovskiy, D. Apalkov, S. Watts, R. Chepulskii, R. S. Beach, A. Ong, X. Tang, A. Driskill-Smith, W. H. Butler, P. B. Visscher, D. Lottis, E. Chen, V. Nikitin, and M. Krounbi. “Basic principles of STT-MRAM cell operation in memory arrays”. en. In: *Journal of Physics D: Applied Physics* 46.7 (Jan. 2013). Publisher: IOP Publishing, p. 074001. ISSN: 0022-3727. DOI: 10.1088/0022-3727/46/7/074001.

[45] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. “Phase Change Memory”. In: *Proceedings of the IEEE* 98.12 (Dec. 2010), pp. 2201–2227. ISSN: 1558-2256. DOI: 10.1109/JPROC.2010.2070050.

[46] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. “The missing memristor found”. en. In: *Nature* 453.7191 (May 2008). Publisher: Nature Publishing Group, pp. 80–83. ISSN: 1476-4687. DOI: 10.1038/nature06932.

[47] L. Chua. “Memristor-The missing circuit element”. In: *IEEE Transactions on Circuit Theory* 18.5 (Sept. 1971). Conference Name: IEEE Transactions on Circuit Theory, pp. 507–519. ISSN: 2374-9555. DOI: 10.1109/TCT.1971.1083337.

[48] L. Chua and S. M. Kang. “Memristive devices and systems”. In: *Proceedings of the IEEE* 64.2 (Feb. 1976), pp. 209–223. ISSN: 1558-2256. DOI: 10.1109/PROC.1976.10092.

[49] S. Kannan, J. Rajendran, R. Karri, and O. Sinanoglu. “Sneak-path Testing of Memristor-based Memories”. In: *2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems*. ISSN: 2380-6923. Jan. 2013, pp. 386–391. DOI: 10.1109/VLSID.2013.219.

[50] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai. “Metal–Oxide RRAM”. In: *Proceedings of the IEEE* 100.6 (June 2012), pp. 1951–1970. ISSN: 1558-2256. DOI: 10.1109/JPROC.2012.2190369.

[51] Y. Biyani, R. Bishnoi, T. Spyrou, and S. Hamdioui. “C3CIM: Constant Column Current Memristor-Based Computation-in-Memory Micro-Architecture”. In: *2025 Design, Automation & Test in Europe Conference (DATE)*. ISSN: 1558-1101. Mar. 2025, pp. 1–7. DOI: 10.23919/DATE64628.2025.10992950.

[52] H. Aziza, M. Fieback, S. Hamdioui, H. Xun, and M. Taouil. “Conductance variability in RRAM and its implications at the neural network level”. In: *Microelectronics Reliability* 166 (Mar. 2025), p. 115594. ISSN: 0026-2714. DOI: 10.1016/j.microrel.2025.115594.

[53] A. Grossi, E. Nowak, C. Zambelli, C. Pellissier, S. Bernasconi, G. Cibrario, K. El Hajjam, R. Crochemore, J. Nodin, P. Olivo, and L. Perniola. “Fundamental variability limits of filament-based RRAM”. In: *2016 IEEE International Electron Devices Meeting (IEDM)*. ISSN: 2156-017X. Dec. 2016, pp. 4.7.1–4.7.4. DOI: 10.1109/IEDM.2016.7838348.

[54] C. Acal, D. Maldonado, A. Cantudo, M. B. González, F. Jiménez-Molinos, F. Campabadal, and J. B. Roldán. “Variability in HfO 2 -based memristors described with a new bidimensional statistical technique”. en. In: *Nanoscale* 16.22 (2024). Publisher: Royal Society of Chemistry, pp. 10812–10818. DOI: 10.1039/D4NR01237B.

[55] J. B. Roldán, E. Miranda, D. Maldonado, A. N. Mikhaylov, N. V. Agudov, A. A. Dubkov, M. N. Koryazhkina, M. B. González, M. A. Villena, S. Poblador, M. Saludes-Tapia, R. Picos, F. Jiménez-Molinos, S. G. Stavrinides, E. Salvador, F. J. Alonso, F. Campabadal, B. Spagnolo, M. Lanza, and L. O. Chua. “Variability in Resistive Memories”. en. In: *Advanced Intelligent Systems* 5.6 (2023). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/aisy.202200338, p. 2200338. ISSN: 2640-4567. DOI: 10.1002/aisy.202200338.

[56] H. Aziza, J. Postel-Pellerin, H. Bazzi, P. Canet, M. Moreau, V. D. Marca, and A. Harb. “True Random Number Generator Integration in a Resistive RAM Memory Array Using Input Current Limitation”. In: *IEEE Transactions on Nanotechnology* 19 (2020), pp. 214–222. ISSN: 1941-0085. DOI: 10.1109/TNANO.2020.2976735.

[57] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. “DaDianNao: A Machine-Learning Supercomputer”. In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. ISSN: 2379-3155. Dec. 2014, pp. 609–622. DOI: 10.1109/MICRO.2014.58.

[58] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. “Architecting Efficient Interconnects for Large Caches with CACTI 6.0”. In: *IEEE Micro* 28.1 (Jan. 2008), pp. 69–79. ISSN: 1937-4143. DOI: 10.1109/MM.2008.2. URL: https://ieeexplore.ieee.org/document/4460514 (visited on 08/15/2025).

[59] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams. “Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication”. In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2016, pp. 1–6. DOI: 10.1145/2897937.2898010.

[60] *Synopsys | EDA Tools, Semiconductor IP & Systems Verification*. en. URL: https://www.synopsys.com/ (visited on 06/23/2025).

[61] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally. “A detailed and flexible cycle-accurate Network-on-Chip simulator”. In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2013, pp. 86–96. DOI: 10.1109/ISPASS.2013.6557149.

[62] A. B. Kahng, B. Lin, and S. Nath. “ORION3.0: A Comprehensive NoC Router Estimation Tool”. In: *IEEE Embedded Systems Letters* 7.2 (June 2015), pp. 41–45. ISSN: 1943-0671. DOI: 10.1109/LES.2015.2402197.

[63] H. Mori, W.-C. Zhao, C.-E. Lee, C.-F. Lee, Y.-H. Hsu, C.-K. Chuang, T. Hashizume, H.-C. Tung, Y.-Y. Liu, S.-R. Wu, K. Akarvardar, T.-L. Chou, H. Fujiwara, Y. Wang, Y.-D. Chih, Y.-H. Chen, H.-J. Liao, and T.-Y. J. Chang. “A 4nm 6163-TOPS/W/b 4790−TOPS/mm2/b SRAM Based Digital-Computing-in-Memory Macro Supporting Bit-Width Flexibility and Simultaneous MAC and Weight Update”. In: *2023 IEEE International Solid-State Circuits Conference (ISSCC)*. ISSN: 2376-8606. Feb. 2023, pp. 132–134. DOI: 10.1109/ISSCC42615.2023.10067555.

[64] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi. “NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.7 (July 2012), pp. 994–1007. ISSN: 1937-4151. DOI: 10.1109/TCAD.2012.2185930.

[65] M. Poremba and Y. Xie. “NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories”. In: *2012 IEEE Computer Society Annual Symposium on VLSI*. ISSN: 2159-3477. Aug. 2012, pp. 392–397. DOI: 10.1109/ISVLSI.2012.82.

[66] L. Steiner, M. Jung, F. S. Prado, K. Bykov, and N. Wehn. “DRAMSys4.0: An Open-Source Simulation Framework for In-depth DRAM Analyses”. en. In: *International Journal of Parallel Programming* 50.2 (Apr. 2022), pp. 217–242. ISSN: 1573-7640. DOI: 10.1007/s10766-022-00727-4.

[67] H. Luo, Y. C. Tuğrul, F. N. Bostancı, A. Olgun, A. G. Yağlıkçı, and O. Mutlu. *Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator*. arXiv:2308.11030 [cs]. Nov. 2023. DOI: 10.48550/arXiv.2308.11030.

[68] *Questa One Sim*. en. URL: https://eda.sw.siemens.com/en-US/ic/questa-one/simulation/questa-one-sim/ (visited on 06/23/2025).

[69] *Veripool*. URL: https://www.veripool.org/verilator/ (visited on 06/23/2025).

[70] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. “The gem5 simulator”. en. In: *ACM SIGARCH Computer Architecture News* 39.2 (May 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.

[71] A. Pullini, D. Rossi, I. Loi, A. Di Mauro, and L. Benini. “Mr. Wolf: A 1 GFLOP/s Energy-Proportional Parallel Ultra Low Power SoC for IOT Edge Processing”. In: *ESSCIRC 2018 - IEEE 44th European Solid State Circuits Conference (ESSCIRC)*. ISSN: 1930-8833. Sept. 2018, pp. 274–277. DOI: 10.1109/ESSCIRC.2018.8494247.

[72] R. Pelke, F. Staudigl, N. Thomas, M. Hossein, N. Bosbach, J. Cubero-Cascante, R. Leupers, and J. M. Joseph. “The show must go on: a reliability assessment platform for resistive random access memory crossbars”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 383.2288 (Jan. 2025). Publisher: Royal Society, p. 20230387. DOI: 10.1098/rsta.2023.0387.

[73] M. A. Yaldagard, S. Diware, R. V. Joshi, S. Hamdioui, and R. Bishnoi. “Read-disturb Detection Methodology for RRAM-based Computation-in-Memory Architecture”. en. In: *2023 IEEE 5th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. Hangzhou, China: IEEE, June 2023, pp. 1–5. ISBN: 979-8-3503-3267-4. DOI: 10.1109/AICAS57966.2023.10168638.

[74] M. Lanza, R. Waser, D. Ielmini, J. J. Yang, L. Goux, J. Suñe, A. J. Kenyon, A. Mehonic, S. Spiga, V. Rana, S. Wiefels, S. Menzel, I. Valov, M. A. Villena, E. Miranda, X. Jing, F. Campabadal, M. B. Gonzalez, F. Aguirre, F. Palumbo, K. Zhu, J. B. Roldan, F. M. Puglisi, L. Larcher, T.-H. Hou, T. Prodromakis, Y. Yang, P. Huang, T. Wan, Y. Chai, K. L. Pey, N. Raghavan, S. Dueñas, T. Wang, Q. Xia, and S. Pazos. “Standards for the Characterization of Endurance in Resistive Switching Devices”. In: *ACS Nano* 15.11 (Nov. 2021). Publisher: American Chemical Society, pp. 17214–17231. ISSN: 1936-0851. DOI: 10.1021/acsnano.1c06980.

[75] R. Pelke, F. Staudigl, N. Thomas, N. Bosbach, M. Hossein, J. Cubero-Cascante, L. B. Poehls, R. Leupers, and J. M. Joseph. “A Fully Automated Platform for Evaluating ReRAM Crossbars”. In: *2024 IEEE 25th Latin American Test Symposium (LATS)*. ISSN: 2373-0862. Apr. 2024, pp. 1–6. DOI: 10.1109/LATS62223.2024.10534593.

[76] L. Geiger and P. Team. “Larq: An Open-Source Library for Training Binarized Neural Networks”. en. In: *Journal of Open Source Software* 5.45 (Jan. 2020), p. 1746. ISSN: 2475-9066. DOI: 10.21105/joss.01746.

[77] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. arXiv:1912.01703 [cs]. Dec. 2019. DOI: 10.48550/arXiv.1912.01703.

[78] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. ISSN: 2575-7075. June 2018, pp. 2704–2713. DOI: 10.1109/CVPR.2018.00286.

[79] B. Parhami. *Computer arithmetic: algorithms and hardware designs*. English. 2nd ed. Oxford series in electrical and computer engineering. New York: Oxford University Press, 2010. ISBN: 978-1-61344-545-7.

[80] *CUDA C++ Programming Guide — CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 08/08/2025).

[81] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo. “UNPU: An Energy-Efficient Deep Neural Network Accelerator With Fully Variable Weight Bit Precision”. In: *IEEE Journal of Solid-State Circuits* 54.1 (Jan. 2019), pp. 173–185. ISSN: 1558-173X. DOI: 10.1109/JSSC.2018.2865489.

[82] L. Roeder. *lutzroeder/netron*. original-date: 2010-12-26T12:53:43Z. Aug. 2025. URL: https://github.com/lutzroeder/netron (visited on 08/01/2025).