Computation-in-memory from application-specific to programmable designs based on memristor devices

Zahedi, M.Z.

DOI
10.4233/uuid:e0a6d2e3-6ec6-4edc-8e6a-5ae931d7dffb

Publication date
2023

Document Version
Final published version

Citation (APA)
Zahedi, M. Z. (2023). *Computation-in-memory from application-specific to programmable designs based on memristor devices*. [Dissertation (TU Delft), Delft University of Technology].
https://doi.org/10.4233/uuid:e0a6d2e3-6ec6-4edc-8e6a-5ae931d7dffb

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

COMPUTATION-IN-MEMORY FROM APPLICATION-SPECIFIC TO PROGRAMMABLE DESIGNS

BASED ON MEMRISTOR DEVICES

COMPUTATION-IN-MEMORY FROM APPLICATION-SPECIFIC TO PROGRAMMABLE DESIGNS

BASED ON MEMRISTOR DEVICES

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. dr. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op vrijdag 22 December 2023 om 10:00 uur

door

Mahdi ZAHEDI

Master of Science in Electrical Engineering,
Universiteit van Teheran, Teheran, Iran,
geboren te Teheran, Iran.

Dit proefschrift is goedgekeurd door de

promotor: Prof. dr. ir. S. Hamdioui
promotor: Dr. ir. J.S.S.M. Wong

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof. dr. ir. S. Hamdioui,	Technische Universiteit Delft, promotor
Dr. ir. J.S.S.M. Wong,	Technische Universiteit Delft, promotor
Onafhankelijke leden:	
Prof. dr. ir. G. Gaydadjiev	Technische Universiteit Delft
Prof. dr. R.V. Joshi	IBM Watson, USA / Technische Universiteit Delft
Prof. dr. F. Catthoor	IMEC / KU Leuven, Belgium
Prof. dr. H. Amrouch	TU Munich, Germany
Prof. dr. H. Corporaal	TU Eindhoven, NL
Prof. dr. ir. W.A. Serdijn,	Technische Universiteit Delft, reservelid

Keywords: Computation-in-Memory, Memristor, Computer Architecture, Hardware Design, Instruction Set Architecture
Printed by: Ipskamp Printing, the Netherlands
Front & Back: By M.Zahedi.

ISBN 978-94-6366-789-0

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

Dedicated to my parents and my lovely sister

CONTENTS

SUMMARY

Computation-in-Memory (CIM) is a promising alternative to traditional computing systems where the storage is conceptually separated from the computing units. Instead, the CIM paradigm aims to perform the computation where the data resides, alleviating the memory bottleneck and ultimately leading to higher energy efficiency and performance. From the memory technology perspective, memristors, emerging non-volatile memory devices, demonstrate various beneficial characteristics. Although the concept of CIM, in combination with these emerging memory technologies, is in the infancy stage, it shows great potential as a future of computing systems. To further understand and quantify the potential of CIM, more development is required at each abstraction level. In this thesis, we first explore the main potentials for memristor-based computation-in-memory. Then, we study different applications from the CIM perspective to understand different behaviors and patterns of applications and use this knowledge to develop architectural solutions for CIM. Based on that, we study the realization of CIM as a generic and flexible platform at a micro-architecture level.

Investigating the Potential of CIM: In the first step, it is essential to have a complete picture of the key features and trends for memristor-based CIM. In this study, we gather the main information from the literature about CIM for different abstraction levels. Based on that, we provide a generic illustration of a CIM-tile, which most existing works can fit into. This can clarify the available design choices as well as the scopes where more attention is required. Parts of this thesis will focus on developing the different components within a CIM-Tile.

Exploiting CIM for Application-Specific Design: A designer should have a good insight into the applications that have the potential to be executed using a memristor-based CIM system. Each application may have a different data and control flow and may require different features to be enabled in CIM. We study how the applications should be mapped into the memory and analyze what the required components next to the memory array are. In this thesis, we focus on three applications from completely different domains (Neural network, Graph processing, and Bioinformatics). The information and experience achieved from this study are also insightful for further development in a more generic design approach as well.

Architectural Solutions for Efficient and Programmable CIM: At the two ends of the spectrum for CIM design methodology, we can have programmable/generic and application-specific designs. The thesis provides a comprehensive study in both directions. Considering the more generic approach, we focus on the (micro) architectures of a CIM-Tile. We design and implement the required components and introduce an instruction set architecture to bring programmability to a CIM-Tile. Besides, we develop a simulation platform to perform design space exploration, verify the functionality, and measure the efficiency of the potential micro-architecture.

SAMENVATTING

Computation-in-Memory (CIM) is een veelbelovend alternatief voor traditionele computersystemen waarbij de opslag conceptueel gescheiden is van de computereenheden. In plaats daarvan heeft het CIM-paradigma tot doel de berekening uit te voeren waar de gegevens zich bevinden, waardoor het geheugenknelpunt wordt verlicht en uiteindelijk leidt tot hogere energie-efficiëntie en prestaties. Vanuit het perspectief van de geheugentechnologie vertonen memristors, opkomende niet-vluchtige geheugenapparaten, verschillende gunstige eigenschappen. Hoewel het concept van CIM, in combinatie met deze opkomende geheugentechnologieën, nog in de kinderschoenen staat, vertoont het een groot potentieel als toekomst voor computersystemen. Om het potentieel van CIM verder te begrijpen en te kwantificeren, is er meer ontwikkeling nodig op elk abstractieniveau. In dit proefschrift onderzoeken we eerst de belangrijkste mogelijkheden voor op memristor gebaseerde berekeningen in het geheugen. Vervolgens bestuderen we verschillende toepassingen vanuit het CIM-perspectief om verschillende gedragingen en patronen van toepassingen te begrijpen en deze kennis te gebruiken om architectonische oplossingen voor CIM te ontwikkelen. Op basis daarvan bestuderen we de realisatie van CIM als generiek en flexibel platform op micro-architectuurniveau.

Het potentieel van CIM onderzoeken: In de eerste stap is het essentieel om een compleet beeld te hebben van de belangrijkste kenmerken en trends voor op memristor gebaseerde CIM. In deze studie verzamelen we de belangrijkste informatie uit de literatuur over CIM voor verschillende abstractieniveaus. Op basis daarvan geven we een generieke illustratie van een CIM-tegel, waar de meeste bestaande werken in kunnen passen. Dit kan de beschikbare ontwerpkeuzes verduidelijken, evenals de scopes waar meer aandacht nodig is. Delen van dit proefschrift zullen zich richten op het ontwikkelen van de verschillende componenten binnen een CIM-Tile.

CIM benutten voor toepassingsspecifiek ontwerp: Een ontwerper moet een goed inzicht hebben in de toepassingen die het potentieel hebben om te worden uitgevoerd met behulp van een op memristor gebaseerd CIM-systeem. Elke toepassing kan een andere gegevens- en controlestroom hebben en vereist mogelijk dat verschillende functies in CIM worden ingeschakeld. We bestuderen hoe de applicaties in het geheugen moeten worden geplaatst en analyseren wat de benodigde componenten naast de geheugenarray zijn. In dit proefschrift concentreren we ons op drie toepassingen uit totaal verschillende domeinen (neuraal netwerk, grafiekverwerking en bio-informatica). De informatie en ervaringen uit dit onderzoek zijn ook inzichtelijk voor verdere ontwikkeling in een meer generieke ontwerpaanpak.

Architecturale oplossingen voor efficiënte en programmeerbare CIM: Aan de twee uiteinden van het spectrum voor CIM-ontwerpmethodologie kunnen we programmeerbare/generieke en toepassingsspecifieke ontwerpen hebben. Het proefschrift biedt een uitgebreid onderzoek in beide richtingen. Gezien de meer generieke aanpak richten we

ons op de (micro)architecturen van een CIM-Tile. We ontwerpen en implementeren de benodigde componenten en introduceren een instructiesetarchitectuur om programmeerbaarheid aan een CIM-Tile te brengen. Daarnaast ontwikkelen we een simulatieplatform om ontwerpruimteverkenning uit te voeren, de functionaliteit te verifiëren en de efficiëntie van de potentiële microarchitectuur te meten.

ACKNOWLEDGEMENTS

The PhD journey is coming to an end with many ups and downs. I am so happy not just because I am receiving a doctorate diploma but mainly because this journey taught me a lot. If we are lucky to experience (limited) freedom in part of our research, we are forced to trust ourselves and realize there is nothing to lean on except our skills and thoughts. To me, the least important part of this journey is about publications, but the skills we learn last for a lifetime, i.e., how we structure our thoughts, criticize and evaluate them, present them to different audiences, make a plan, and manage our time are the main skills we achieve. This path adds even more if you are an international student starting your life from scratch in a new country where you do not speak its language, you do not know about its culture, and you do not have the support of your family and old friends. However, this journey could not have ended successfully without the support I received during these more than four years. In the following, I would like to express my appreciation to those who helped me to stand where I am now.

First, I would like to thank my promoters, **Prof. dr. ir Said Hamdioui** and **Dr. ir Stephan Wong**. Said, thanks for giving me the opportunity to start my PhD here in Delft. You are very energetic and supportive. You never force for a publication; instead, look for high-quality work. You taught me that presenting your idea is as important as the idea itself. Stephan, you were more than a supervisor for me, especially during the pandemic, which was almost half of my PhD. I found you very caring, considerate, precise, and thoughtful. You were more involved in the technical details of my work, and your feedback and discussion helped me. Once, you told me, "Never fall in love with your ideas", which I believe helps me in all aspects of my life. Thanks for everything, Said and Stephan. I would also extend my appreciation to **Prof. Dr. ir Kees Goossens**. Kees, you were the one who opened up a window to new opportunities and a new world for me. We worked together for a short time, but it was enough for me to realize you are among a few professors who are still passionate about technical details and implementations. Thanks, and I wish you all the best.

Next, I want to thank my academic collaborators, specifically during the MNEMOSENE project. Although the pandemic hit the project, we still had many informative meetings, and we wrote many work packages together. I would like to thank **Prof. dr. Francky Catthoor**, **Prof. dr. Henk Korporal**, **Dr. Abu Sebastian**, **Dr. Manuel Le Gallo**, **Dr. Dirk Wouters**, **Dr. Stephan Menzel**, and **Dr. Fernando García-Redondo**, with whom we had the main collaboration. In addition, I thank **Dr. Rajendra Bishnoi**, **Dr. Motta Taouil** and **Prof.dr.ir Georgi Gaydadjiev**. Rajendra and Motta, we had many nice discussions during our break, and I always enjoyed talking to you. Georgi, although you joined the group at the very end of my Ph.D., I appreciate your comments on my work and hope we collaborate more in the future.

During this journey, I was lucky to have very nice colleagues. The memories we cre-

ated together will always be a source of happiness for me. I would like to thank **Moritz** and **Taha**, who are also my paranymphs. Mortiz, my Dutch friend, thanks for always being supportive. Having such a friend gives international students a feeling of safety. Taha, I feel happy and lucky to have you as a friend. I will not forget how much we laughed and grumbled together. Without you, the office would be dull. I would like to name other colleagues/friends as well: **Abdulghader, Abdullah, Muath, Amin, Troya, Haji, Shayesteh, Mansureh, Erik, Mark, Francis, Trisha, Joyce, Guilherme, Lizhou, Abhairaj, Sumit, Yash, Asmae, Shardul, Folkert, Matti, Michael, Alexandra, Heba, Anteneh, Cezar, Daniel, Oscar, Simon, Fouwad, Arne, Pantazis, Geerten, Christopher, Remon, Geert,** and **Li Oh**.

I should also thank some of my friends in the Netherlands with whom we created fun memories and spent time together: **Alireza, Nima, Shayan, Hadi, Ali, Hossein, Hamid, Sahar,** and **Mohammad**. I will never forget my lovely friends back in Iran who are always ready to talk whenever I need them: **Abbas(G), Hasan (A), Masoud (H), Hossein (S), Amir-Hossein (K)**. I am incredibly grateful to my excellent teacher **Yaser Rastegari** for the continuous efforts he put in for us from secondary school until now to be better human beings.

Finally, I must express my profound gratitude to my parents, my sister, and my uncle, Hamed, for providing me with unfailing support and continuous encouragement throughout these years. I hope we can spend more time together from now on. I love you so much.

Mahdi
Delft, September, 2023

1

INTRODUCTION

Nowadays, computing systems such as embedded systems, personal computers, and servers contribute to human life more than ever, becoming necessary for even daily activities. In the last decades, computing performance was able to keep up with the increasing demands of applications and requirements thanks to transistor downscaling and technological improvement. However, CMOS downscaling is reaching its physical and economic limits. Hence, researchers are investigating novel architectures and new technologies as an alternative or complement to conventional computing systems. In this chapter, we first describe the motivation behind computation-in-memory as a novel architectural solution and emerging memristor devices. Second, we list some opportunities and challenges of such a design. Third, we explain briefly the research direction of this thesis.

1

1.1. MOTIVATION

Modern-day applications such as machine learning, image processing, and encryption are widely employed on real-time embedded systems as well as back-end data centers. The amount of data to be processed for these types of applications has increased considerably (e.g., BERT has around 110 million parameters). The main platforms to execute these applications are based on Von-Neumann architecture, where the storage is separated from the processor. Hence, these large amounts of data are far from the processing units. Consequently, the performance and power consumption of the systems that fulfill these target applications have become crucial. Traditional Von-Neumann-based architectures have been stretched to attain adequate performance at reasonable energy levels, but clearly show limitations for further improvements. One of the main limitations is that transporting data between the processing unit(s) and the memory leads to huge energy consumption and long latencies. This greatly diminishes the usefulness of Von-Neumann computers. Moreover, the limitations of Von-Neumann architecture are compounded by the challenges of CMOS transistor downscaling, one of the main drivers for performance/energy improvement for many years. In the following, we elaborate more on Von-Neumann computers. Subsequently, we discuss the limitations we currently face from the Von-Neumann computers and CMOS transistor downscaling before introducing computation-in-memory using memristor devices.

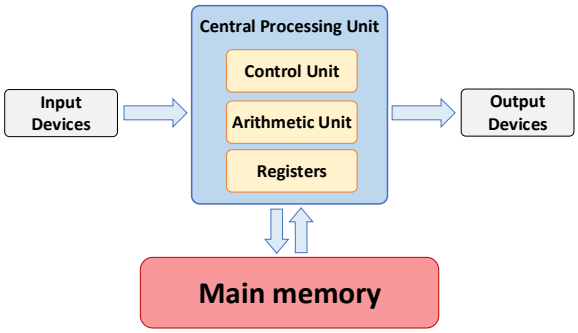


Figure 1.1: Traditional Von-Neumann architecture and conceptual separation of the processing unit and its memory

1.1.1. VON-NEUMANN ARCHITECTURE

In general, data processing (within a computer) entails the transformation of data via computations (performed by processing units). Furthermore, the type and order of computations are controlled by a program. In vN architectures, the data and program (stored in memory) are separated from the processing units. Accordingly, the architecture conceptually consists of three main blocks as depicted in Figure 1.1: the central processing unit (CPU), memory, and input/output devices. These devices produce input data or consume the output data, respectively. The CPU is responsible for executing the instructions of a program and performing logical/arithmetic operations on data. During

the execution of a program, the data and instructions are loaded from the main memory, and the result of the operations is sent back to the memory and/or output devices.

Over the past decades, there has been much work on improving CPUs' performance and energy efficiency. This shifts the bottleneck of computing systems in terms of performance and energy into memory. There are two main challenges with respect to the main memory. First, the cost of data transactions between the main memory and CPU in terms of latency and power is considerable. For instance, communication between off-chip DRAM and the processor takes up to 200 CPU cycles [1], and it consumes power more than two orders of magnitude than a floating-point operation inside the CPU [2, 3]. This resulted in the introduction of a memory hierarchy (see Figure 1.2) that exploits temporal and spatial locality of data to store recently used data (temporal) and data close to recently used ones (spatial). As illustrated in the figure, on top of this hierarchy, modern computers employ small SRAM-based cache memory fabricated in the same chip as the CPU. A key requirement is that the data that is operated on by the CPU must fit within the different caches at the different levels within the hierarchy to deliver a high-performance and energy-efficient system. This leads us to the second challenge concerning memory. Unfortunately, modern-day applications no longer meet this requirement and thereby making the memory hierarchy obsolete or ineffective for the type of data they process. Hence, another layer of hierarchy, secondary memory, has to be employed. The secondary memory is mainly implemented as a flash or a hard disk drive. Moving the data across memory now involves many steps, which adds up to the overhead of data communication, thereby making it even slower. For example, the data transfer between secondary and main memory is managed by an operating system running on the same CPU. The operating system configures a direct memory access (DMA) module to transfer the data and alleviate the intervention of the CPU in such transfers. Modern-day applications should go through all these steps frequently during their execution. It should be clear that the current memory hierarchy paradigm (extended to physical disk) is no longer sufficient for modern applications. In conclusion, new solutions must be sought after to support modern applications.

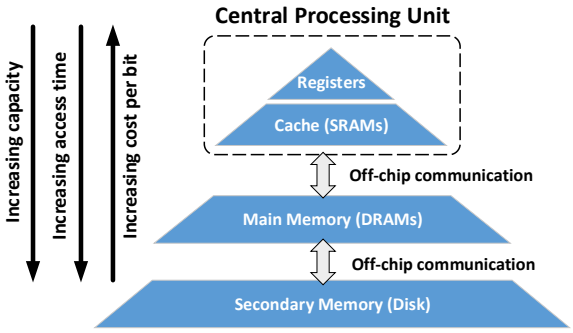


Figure 1.2: Memory hierarchy in traditional processors

1

1.1.2. LIMITATIONS OF TRADITIONAL COMPUTING SYSTEMS

For decades, technology and architectural improvements were two major drivers for delivering high-performance and energy-efficient systems. From the technology perspective, the feature size of transistors has kept scaling down. The scaling theory proposed (Moore's law) was the beginning of the flourishing era of electronic devices. Since then, by scaling the transistor feature size, more and more transistors are integrated into the circuits, and more complex functionalities are implemented. The transistor scaling brought higher energy, performance, and area efficiency for the designs for many years. However, transistor downscaling is now getting close to the end by facing major 'walls'. In addition to technology, architectural improvement also has made a big contribution to the efficiency of computing systems for many years. Similarly, we encountered some walls that limit the improvements we can expect from the architectural techniques. In the following, we elaborate on these walls to provide a complete picture of all major limitations.

1. **Leakage Wall**: As the transistors are scaled down, gate thickness and the length of the channel between the drain and source are reduced as well. Nowadays, this thickness comprises only a few layers of atoms. Hence, the probability for quantum mechanical tunneling growths, which then increases gate leakage current. In addition, shorter channels could cause high off-state drain leakage. Finally, since the supply voltage should be reduced in the lower technology nodes, the threshold voltage is also scaled down to keep the performance the same. However, this reduces the room for gate voltage to swing below the threshold voltage. Hence, the sub-threshold leakage current increases more when a transistor is in the off state.

2. **Reliability Wall**: As the size of transistors scaled down, small deviations due to (fabrication) process variations now have a relatively higher probability to (negatively) influence the transistor's functionality. Moreover, the dielectric and wiring materials are unable to provide reliable insulation and conduction with continued scaling.

3. **Power Wall**: As the size of transistors is scaled down, more transistors can be populated in a fixed area. Table 1.1 shows the number of transistors used for different generations of Intel processors. According to Dennard scaling, the power density of the chip (now with more transistors) stays constant due to voltage and current downscaling. However, due to some other factors, Dennard scaling has stopped. Consequently, no longer possible to drive in the same area with the same amount of power. This means more power is needed if more transistors need to be switched per area. Now, considering the fact that the performance of single-core stagnated, and to explore more parallelism, the industry chose to design multi-core chips, but with a caveat. Since Dennard scaling has stopped, at some point, not all cores can be switched on all the time. The main reason is that extracting the generated heat from chips has become technologically impossible without costly (thus, commercially unviable) measures. Hence, we faced a new phenomenon called 'Dark Silicon,' where more than 50% of the chips are not operational at the same time due to power limitations [4]. For the same reason, we can also observe in Figure 1.3(a) that the clock frequency of the processors, which has a direct relation

with power consumption and the performance of the system, has been saturating in the past few years. These indicate that we can no longer expect significant performance improvement from increasing the number of cores due to the Dark Silicon phenomenon.

4. **Cost Wall**: The design and fabrication cost increases as the transistor feature size is reduced. This is contributed by equipment, lithography process, mask, and test costs.

5. **ILP Wall**: Another source of performance improvement is Instruction Level Parallelism (ILP), where multiple instructions are executed in parallel. This technique can improve the performance until the points where the applications or algorithms inherently have this degree of parallelism. After this point, providing more levels of parallelism in the hardware will not result in significant performance improvement. Besides this limitation factor from applications, the impact of the power wall influences the maximum parallelism that the hardware can enable. Figure 1.3(b) shows the potential performance improvement expected by Moore's law against achieved performance improvement as we downscale the transistor feature size. One of the main reasons for this gap is the ILP wall.

6. **Memory Wall**: As the energy efficiency and performance of processors improved, data communication to and from the processors could not keep up with the same peace. As mentioned before, the off-chip communication imposes an overhead on performance and energy up to two orders of magnitude than a single floating-point operation inside the CPU [2, 3]. The amount of data that can be transferred in parallel is also limited to the available bandwidth. This memory wall is another reason for the gap we observe in Figure 1.3(b). We can conclude that the memory wall has a major impact on a system's efficiency in terms of energy and performance.

Considering those walls, our main drivers for decades to improve energy and performance are no longer available. Hence, finding alternative technologies and architectures is necessary to realize higher efficient designs demanded by emerging big-data applications.

Table 1.1: The number of transistors for different generations of Intel processors.

Intel processors	Transistor Count	Year
Intel 4004	2.300	1971
Intel 8086	29.000	1978
Intel i860	1.000.000	1989
Pentium 1	3.100.000	1993
Pentium 4	112.000.000	2004
Core i7	731,000,000	2008
Quad-core + GPU Core i7	1,160,000,000	2011
Quad-core + GPU Core i7 Ivy Bridge	1,400,000,000	2012
Quad-core + GPU GT2 Core i7 Skylake	1,750,000,000	2015
28-core Xeon Platinum 8180	8,000,000,000	2017

1.1.3. COMPUTATION-IN-MEMORY BASED ON MEMRISTIVE DEVICES

Computation-in-Memory: The memory wall is one major bottleneck for energy and performance improvement of traditional computing systems. Consequently, it is only

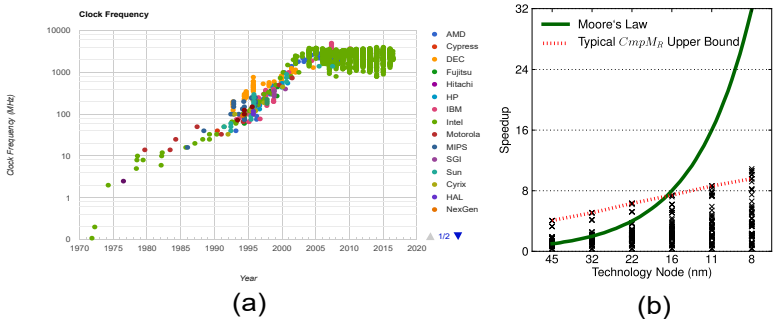


Figure 1.3: (a) Maximum clock frequency for different processors over the past decades [5]. (b) Actual performance increase over different technology nodes for PARSEC benchmarks reported for multicore model (CmpM) versus the expectation based on Moore's law [4]

logical to move the computing unit as close as possible to the data location, which is usually the main memory. The closeness can be categorized into "near-memory" and "in-memory". An example of near-memory computing is 3D stacking technology, in which computation is performed in the DRAM chip by adding extra logic next to the memory structure. As we perform the computation closer to the storage, we reduce the cost of data transfer in terms of performance and energy, and we are expecting to have even higher bandwidth. Hence, recently, computation-in-memory (CIM), as a concept, has gained a huge interest from the research community in combining memory storage and actual computing in the same memory array. Researchers are exploring this concept for both traditional memory technologies like SRAM, DRAM, and flash memories, as well as emerging memory technologies.

Memristive Devices: From the technological perspective, memristive devices have the potential to replace traditional memories in order to address or alleviate some of the aforementioned walls. The following briefly explains the memristor devices and gives a general feeling of where these devices stand compared to the leading conventional technologies, SRAM, DRAM, and Flash. More information is provided in Chapter 2. The mainstream memristive technologies are Phase Change Memory (PCM), Spin-Transfer Torque Random Access Memory (STT-MRAM), and Resistive Random access memory (ReRAM). These memories represent data as different resistance levels rather than the absence or presence of charge used in conventional memories. Hence, they are non-volatile in contrast to SRAM-based cache and DRAM-based main memory. This reduces the static power of the memory (Leakage wall). Regarding area, memristors are smaller than SRAM, but the same size as DRAM or flash memories. However, since some memristor technologies allow storing several bits per single device, their density (area efficiency) can potentially surpass DRAM and flash memory. In terms of latency, these devices are comparable to DRAM, but quite faster than flash memory. Another important factor for memory technologies is programming/write energy. Although memristors are quite behind SRAM and DRAM, they are still superior to Flash memories [6–8].

In this thesis, we focus on computation-in-memory based on emerging memristor devices due to their unique characteristics mentioned before.

1.2. CHALLENGES OF MEMRISTOR-BASED CIM

Although computation-in-memory and memristor devices can potentially bring considerable energy and performance improvement to the system, some challenges and open questions may arise. They need to be explored and addressed to make CIM a reality. Figure 1.4 lists some of the challenges and opportunities that are potential research domains for CIM. In the following, we briefly describe some of them for each abstraction level.

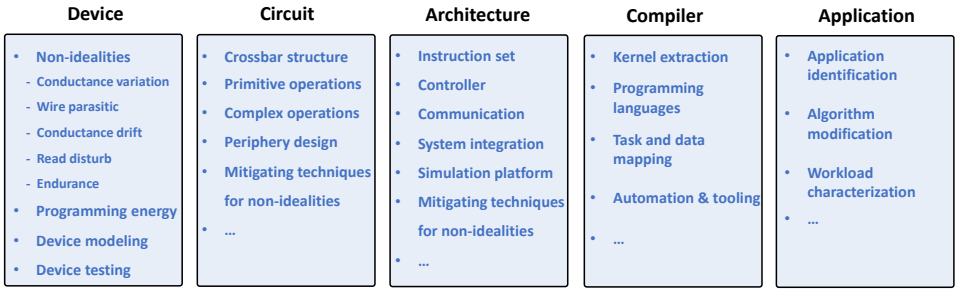


Figure 1.4: Example of challenges and open research directions for computation-in-memory in different abstraction levels.

1.2.1. DEVICE LEVEL

There is much ongoing research in order to explore different materials and technologies to implement memristor devices. A significant focus of research at this level is addressing the non-idealities of the devices. As depicted in Figure 1.4, variation in conductance both during the programming or read phase (drift) is one of the major challenges. A severe conductance drift can lead to read disturbance, which means the state of the device is changed after a few read operations. This gets even more challenging during the computing phase, where this effect is aggregated from different devices. More information can be found in [9] (❶ non-idealities). Besides non-idealities, another concern is the high programming energy of these devices. Reducing this energy is another research challenge (❷ programming energy). In addition, creating a model of these devices to be used for circuit simulations is an active study [10] (❸ device modeling). Finally, finding a proper method to test these devices based on their unique defects is another challenge [11] (❹ device testing).

1.2.2. CIRCUIT LEVEL

At this level, there are also challenges, mainly regarding how the memristors should be connected to form a memory array and how the periphery circuits should be designed to improve the system's desired metrics. The way the computation is performed depends on how the memristor devices are connected to each other. Different memory array structures can enable different operations and impact different system metrics [12] like energy and accuracy (❶ crossbar structure). Furthermore, the researchers explore enabling more primitive and complex operations in memory [13]. Primitive operations (e.g., logical operations, VMM) are the building block for complex operations (❷ primitive/complex operations). Besides array structure and operations, many analog and digital periphery circuits are required to exploit computation in a memory array, and they significantly impact the energy/performance efficiency of the system [14] (❸ periphery design). Finally, finding techniques to mitigate the effect of device non-idealities is another research topic [15] (❹ mitigation techniques for non-idealities). Circuit design for CIM is in the early stages, and more research has to be conducted to find optimum design choices.

1.2.3. ARCHITECTURE LEVEL

Compared to the device- and circuit-level works, there has been less focus on architecture and higher abstraction levels. Some aspects should be studied when an in-memory memristor-based architecture has to be designed. Each memristor tile may consist of different components. A designer should study a tile's main components, how they are controlled, and how much flexibility is required for the control system. The system can deliver a static function or be programmed using a set of ISA [14, 16] (❶ instruction set and controller). Real-world applications need to comprise many memristor tiles. Architects should study communication networks based on the system's requirement to orchestrate data among memory arrays efficiently [17] (❷ communication). The researchers should also study how these memory arrays are integrated into a system and how they should interface with the outside (❸ system integration). Finally, as there might be many parameters in the system, it is essential to develop a simulation platform to realize the system's behavior in different scenarios [18, 19] (❹ simulation).

1.2.4. COMPILER LEVEL

As described before, a circuit designer can enable certain operations in the memory arrays. Based on these operations, we should be able to extract potential parts from an application to be performed inside the memory [20] (❶ kernel extraction). To easily identify these parts, the application should follow specific rules and standards. Then, the selected parts should be decomposed into sets of understandable operations and instructions for the memory [21] (❷ programming language). Besides, how data should be mapped into the memory with respect to the requirements and constraints can considerably influence the system's metric. For example, good mapping can minimize communication among memory arrays [16] (❸ task and data mapping). Finally, a toolchain should be developed to automize these steps (❹ automation). Until now, as CIM is in its early stage, limited attention is paid to the compiler and programming language.

1.2.5. APPLICATION LEVEL

Despite general-purpose processors, computation in memory can be performed under specific conditions mainly due to the limited supported operations and how they should perform. In addition, to exploit maximum efficiency from CIM, the computations should be able to be performed in a massively parallel way. Therefore, we have to evaluate different applications to perceive whether they have the potential to be accelerated by CIM [22, 23] (❶ application identification). In some cases, by modifying an algorithm, we can optimize it for CIM. Hence, an application expert can provide recommendations to alter algorithms and make them compatible with CIM [24]. This is a helpful insight for researchers working on compiler and mapping abstraction level (❷ algorithm modification). Besides, valuable insight can be provided for architects and circuit designers to optimize control flow, data flow, and essential components around the memory arrays based on the application behavior. These characterizations can even influence device-level abstraction. As an example, an application expert can identify how much inaccuracy a device can inject into a system still to satisfy the final accuracy requirements for different applications (❸ workload characterization). In conclusion, studying different applications is essential to enabling CIM and realizing its merit.

1.3. RESEARCH TOPICS

The previous sections have described the potential of memristive devices and computing in memory. However, still many challenges mentioned in Section 1.2 have to be addressed to bring CIM into reality. This thesis mainly focuses on architecture and application levels and aims to propose solutions for some of their challenges. Many researchers in this domain propose different accelerators specifically tailored for an application under certain assumptions and constraints. These studies can clearly show the potential of memristor-based CIM and can provide a good understanding of CIM, its capabilities, and its limitations. Although these accelerators improve the efficiency of computing systems, they become obsolete quickly as the applications change or even evolve. As depicted in Figure 1.5, this imposes a considerable design cost. In this thesis, while we still think it is worth investigating different applications and designing memristor-based accelerators for them, we also study a second path where we give flexibility and programmability to the designs to extend the range of their applications. The thesis provides a comprehensive study in both directions, 1) Application-specific design approach and 2) Generic design approach.

In short, the research carried out in this thesis expects to find an answer to the following question:

What are the main potentials for memristor-based computation-in-memory, and how to realize them for generic and application-specific designs to achieve high energy efficiency and performance?

In the following, we elaborate more on the research question and determine the scope of the thesis with respect to Figure 1.4.

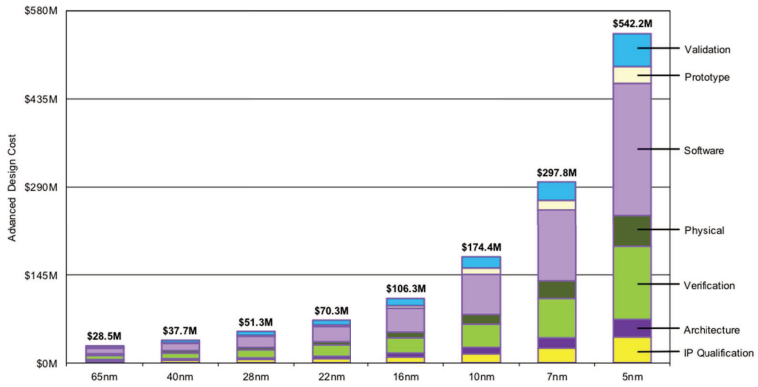


Figure 1.5: Chip design cost under different process nodes [25].

1.3.1. INVESTIGATING THE POTENTIAL OF CIM

During the last few years, many works have been carried out to promote the capabilities of CIM. We will present several key achievements in the memristor research that mainly define low-level functionalities combined with high-level abstractions to estimate the performance and energy (saving) potential. This leaves a large gap that needs to be explored. In our investigation, coinciding with research at the tile level, we determined that functionalities within memristor tiles can be categorized according to a model we developed. Using our model, we can clarify the available design choices as well as the scopes where more attention is required. This information can help to identify the possible future directions.

1.3.2. EXPLOITING CIM FOR APPLICATION-SPECIFIC DESIGN

A designer should have a good insight into the applications that have the potential to be executed using a memristor-based CIM system. Each application may have a different data and control flow and may require different features to be enabled in CIM. Hence, to maximize efficiency for each application, we studied how the applications should be mapped into the memory and whether we need to modify the algorithm to make it suitable for CIM. In this thesis, we studied three applications from completely different domains (Neural network, Graph processing, and Bioinformatics). Afterward, we designed and proposed an energy-efficiency and high-performance memristor-based accelerator for each. The information and experience achieved from this study are also insightful for further development in the subsequent part of the thesis.

1.3.3. ARCHITECTURAL SOLUTIONS FOR EFFICIENT AND PROGRAMMABLE CIM

Considering the generic direction, we focus on the (micro) architectures of a generic CIM-Tile. To realize a generic CIM-Tile model, we studied the common essential com-

ponents required next to a memory array, how the data flows, and what the possible approaches are to orchestrate these components. It is also essential to comprehend how a memory array interfaces with the outside and how the data flows in and out. The way the components next to a memory array are organized and controlled can influence the efficiency of CIM. Poor architectural implementations can lead to low efficiency. Hence, in this part of the thesis, we focus on an efficient and flexible structure of a CIM tile, introduce our instruction set architecture, and design some essential periphery components (e.g., controller) of a generic CIM. Figure 1.6 highlights the parts in a CIM-tile that this thesis focuses on concerning this design direction. Finally, we need to develop a simulation platform to verify the functionality and measure the efficiency of the potential architecture.

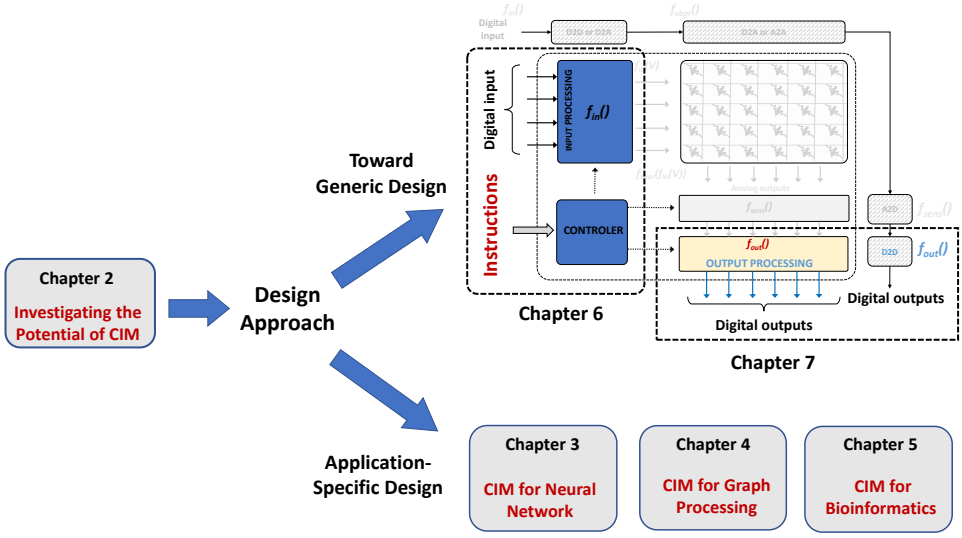


Figure 1.6: Thesis storyline with respect to the thesis chapters

1.4. THESIS CONTRIBUTIONS

The contributions of this thesis are directly related to the research topics discussed in the previous section. The research topic consists of three parts: 1) Investigating the potential of CIM, 2) Exploiting CIM for Application-Specific Design, and 3) Architectural solutions for efficient and programmable CIM. We categorized the contributions of this thesis based on that.
(1) Investigating the potential of CIM:

- **Exploring potential architectural directions and applications for CIM:** We demonstrate the CIM concept using a broader and generalized model. Considering this model, the state-of-the-art CIM-based logic and arithmetic primitive functions,

which can be the building blocks for complex functions, are investigated. Besides, we present potential applications of CIM, which provides insights into the challenges and opportunities of a generic CIM system design. Finally, we highlight the future directions regarding the construction of CIM-based systems. This work has been published in [13].

(2) Exploiting CIM for Application-Specific Design:

- **Designing a high-performance accelerator for binary neural networks:** Neural Networks (NNs) are leveraged in a variety of applications. With the growth of the network size for advanced applications, implementing NNs has become challenging, considering hardware limitations. In this study, we focused on Binary Neural Networks (BNN) due to their high model compression rate and simplified computations. This work proposes an efficient implementation of XNOR-based BNN to maximize parallelization. In this implementation, costly analog-to-digital converters are replaced with sense amplifiers with custom reference(s) to generate activation values. Besides, a novel mapping is introduced to minimize the overhead of data communication between convolution layers mapped to different memristor crossbars. This work can be found in [24].

- **Presenting a novel data representation and accelerator for sparse graphs:** Graph processing is employed in a wide range of areas, including but not limited to social media analysis, urban planning, and machine learning. Due to the explosion of graph size and massive data movement across the memory hierarchy, it is worth exploring the potential of CIM in this application. In this work, we propose a novel data representation tailored for spars-based graph processing and targeting computing-in-memory designs. This enables the computations over a compressed graph representation. The suggested hierarchical mapping, as well as its implementation, can significantly improve the resource utilization of the system, which can lead to considerable energy efficiency. This work has been published in [22].

- **Designing Energy-Efficient food profiler using CIM:** Bioinformatics is another application domain that often works on massive data structures. Hence, it is essential to investigate this domain and its applications as a probable candidate who can exploit CIM. Food profiling is an essential step in any food monitoring system needed to prevent health risks and potential fraud in the food industry. In this study, we propose an accelerated leveraging of the concept of hyperdimensional computing (HDC) and CIM together. This work actualizes several domain-specific optimizations and exploits the inherent characteristics of memristors to improve the overall performance and energy consumption of food profilers. This work has been published in [23].

(3) Architectural solutions for efficient and programmable CIM:

- **Proposing generic in-memory ISA and CIM architectures:** We define a new set of ISA capable of accurately reflecting and flexibly controlling all functional aspects of a generalized CIM-tile organization while dealing with different constraints,

configurations, and requirements. The introduced ISA bridges the gap between high-level programming languages and the CIM-tile architecture and allows for the definition of a compiler. Based on that, the architecture of a CIM tile, which is a generalization of the existing works, is designed. This work has been published in [14].

- **Developing a simulator and compiler for CIM:** We develop a compiler that is able to translate higher-level operations (e.g., defined in higher-level programming languages) into a sequence of instructions in our ISA based on the system configuration. Besides, in order to mimic the behavior of the system, we developed a fully parameterized simulator that is capable of executing the newly introduced instructions and simulating our CIM tile architecture. This work has been published in [21].

- **Suggesting an energy-efficient periphery structure to support unsigned integer MMM:** In order to support the unsigned integer datatype, we propose a new adder structure in the periphery next to the memory crossbar. The proposed design utilizes minimum-sized adders customized based on technology-driven restrictions, e.g., the number of active crossbar rows per addition. The structure is not restricted to a single fixed integer word size and can flexibly support different word sizes without hardware changes. This work has been published in [26].

- **Proposing a novel scheme to support signed arithmetic operations for CIM:** To broaden the scope of CIM and make it applicable to a wide range of applications, different data types should be supported. Many applications are often operating on signed numbers. Hence, without the support of signed numbers, the promise of CIM for these applications is greatly diminished. Therefore, it is essential for researchers to enable this feature for CIM-based design. This work advances the state-of-the-art by proposing a novel mapping solution to support signed and unsigned MMM based on widely used two's complement representation. This method performs signed and unsigned operations with minimum energy, latency, and area overhead by eliminating the costly sign extension. This work has been published in [27].

1.5. THESIS ORGANIZATION

The organization of the thesis is illustrated in Figure 1.6. In the following, we give an overview of the thesis per chapter.

Chapter 2 presents the overview of memristor devices and state-of-the-art CIM designs. First, it presents the background on resistive devices, current operations supported in the memristive crossbar array, and reputed architectures. Second, we provide an overview of applications for CIM. Finally, we briefly discuss the possible architectural choices.

Chapter 3 explores the first application, binary neural networks (BNN), for CIM. We focus on application-specific design for BNNs. In this chapter, we first provide a background on BNNs and their XNOR-based implementation. Afterward, we propose our

implementation, replacing analog-to-digital converters (ADC) with sense amplifiers (SA) using adjusted reference(s). Then, we discuss how the data should be mapped in order to reduce the cost of data communication among different layers of the network. Thereafter, we study the effect of references for SAs on the accuracy of the network, and we evaluate the energy and performance of the system.

Chapter 4 studies CIM in the context of graph processing applications. Due to the considerably large datasets in this domain, these applications are worth to be considered for CIM. In this chapter, we propose an accelerator based on a hierarchical mapping of graph data into the crossbar to improve the energy efficiency of sparse-based graph datasets on the CIM crossbars.

Chapter 5 focuses on another application domain, bioinformatics, which has great potential to leverage the concept of CIM. We first provide background about metagenomic profilers and, specifically, food profilers. The food profiling problem can be expressed as a multi-object (multispecies) classification where hyper-dimensional computing (HDC) can be used. Next, a brief background about HDC is given. Afterward, we propose our design using both CIM and HDC to implement a fast and energy-efficient food profiler.

Chapter 6 explains the architecture of the CIM tile memory array. The CIM tile is controlled by a set of instruction set. Hence, we discuss the instructions and what they do in the CIM tile. Afterward, we present the current version of the compiler and how the instruction sequences are generated. Furthermore, we describe our SytemC CIM tile simulator. Then, we touch on the concept of multi-tiling and briefly describe our idea of inter-tile communication. In the end, we describe our first step toward system integration.

Chapter 7 first describes the digital periphery design to support integer arithmetic operations. This periphery has to take the intermediate results computed in the crossbar and produces the final result. The proposed design flexibly supports different data sizes while minimizing the required hardware and maximizing performance and energy efficiency. Second, by introducing the concept of virtual bit-line and virtual input segment, signed arithmetic operations using two's complement representation are supported in the crossbar with minimum energy and performance overhead. This is obtained by slight modification in the controller of the crossbar.

Chapter 8 concludes this thesis and presents insights on future work.

2

Background and Classification

This chapter presents a background on computation-in-memory based on memristor devices. First, we explain the theory of memristor devices and their three main technologies. We compare these emerging memory devices with mainstream memory technologies and elaborate more on their pros and cons. Subsequently, we present the main crossbar structures to connect these devices together. Afterward, we provide a generic illustration of a memristor tile, comprised of a crossbar and its peripheries, and classify all kinds of operations we can perform in it. Then, we present an overview of potential applications for CIM and different design strategies we may take to implement it. Finally, the chapter concludes with an overview of existing designs and accelerators that exploit CIM and memristor devices.

This chapter is partially based on [13].

Memory is the core part of computing systems, and traditionally, it uses solely to store information; the information is stored as the presence or absence of charge, such as static random access memory (SRAM), dynamic random access memory (DRAM), and flash memories. Besides traditional memories, there is also a new class of memories where the data is stored as resistance (conductance) levels. This memory class refers as the memristive device due to manifesting particular relation between electrical parameters. In the following, we elaborate more on this.

2.1. MEMRISTIVE DEVICE THEORY

Before 1970, the three fundamental circuit elements, resistance(R), capacitance (C), and inductance (L) were known and implemented. These elements relate four fundamental units of electricity (charge, current, voltage, and flux) to each other. The resistance relates the derivative of voltage to current ($dv = Rdi$). The capacitance shows the relation between charge and voltage ($dq = cdv$). Finally, the inductance demonstrates the relation between flux and current ($d\Phi = Ldi$). This is illustrated in Figure 2.1. In 1971, Leon Chua identified the characteristics of a fourth fundamental electrical element that links magnetic flux and charge. This element is called memristor. Equation 2.1 presents the behavior of this element mathematically. Despite resistor (R), memristor provides a dynamic relation between current and voltage which means their present behavior is determined by the voltage, current, and its state.

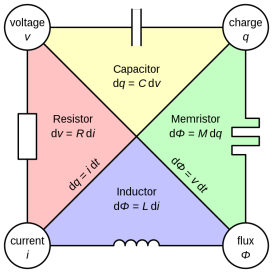


Figure 2.1: Illustration of four fundamental circuit elements and how they relate to each other.

$$M(q) = d\Phi/dq => M(q(t)) = (d\Phi/dt)/(dq/dt) = V(t)/I(t) \tag{2.1}$$

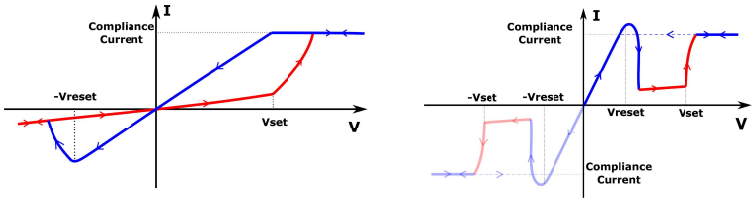
Equation 2.2 shows a more simplified representation of the memristor device. This equation model the memristor as state-dependent ohm's law, where the resistance is a function of state variable x, voltage, and current. In this equation, R represents the resistance function of the memristive device. The state itself is obtained from Equation 2.3, where the time derivative of the state variable is a function of the state variable, voltage, and current. In 2008, HP lab reported the first physical demonstration of the memristor. The memristive device is implemented as a two-terminal device that exhibits a pinched

hysteresis at the origin of the current-voltage (I-V) curve. This pinched hysteresis represents the resistance switching depending on the history of the resistance and the applied voltage or current. This means we can change the resistance state by a change in the voltage (or current).

$$V = R(x, V, I)I \tag{2.2}$$

$$dx/dt = f(x, V, I) \tag{2.3}$$

Figure 2.2 demonstrates the switching dynamics of (a) bipolar and (b) unipolar memristors. In bipolar devices, by applying positive and negative voltage, we can alternate between low and high resistance levels, respectively. This is shown in Figure 2.2(a). Bipolar switching has been demonstrated by several RRAM [28–38] and MRAM [39–43] devices. However, unipolar devices depend only on the magnitude of the programming voltage regardless of its polarity. The unipolar switching has been demonstrated by some RRAM [44–48], and PCRAM devices [49–51].



(a) Bipolar switching (b) Unipolar switching

Figure 2.2: Memristor switching dynamics for (a) bipolar and (b) unipolar devices.

2.2. MEMRISTIVE DEVICE TYPES

Different technologies and materials can be used to implement memristor devices. In the following, we briefly discuss three leading technologies of memristor devices, and then we compare them with conventional memory technologies.

2.2.1. RESISTIVE RANDOM-ACCESS MEMORY

Resistive Random-Access Memory (ReRAM or RRAM) devices consisting of a metal-insulator-metal stack; the bipolar device is set and reset by changing the polarity of the programming voltage (e.g., 2V) to form or dissolve the conducting filament. To read the device without disturbance, a small voltage (e.g., 0.2V) is applied, and the current (voltage) through (across) the device should be sensed while programming the device requires higher voltage/current and longer latency [21]. The device in LRS and HRS states is illustrated in Figure 2.3(a).

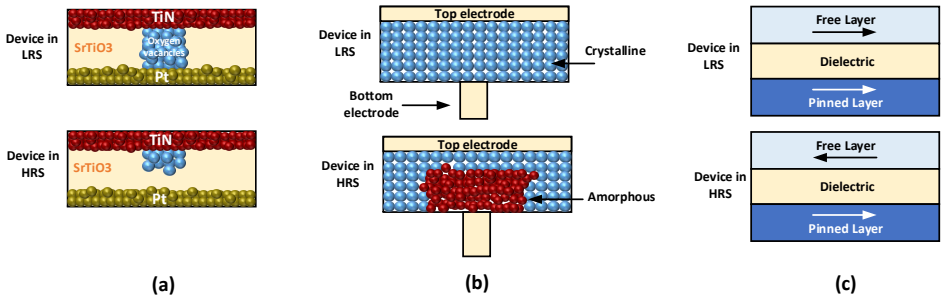


Figure 2.3: Illustration of (a) ReRAM device. The appropriate voltage levels form or dissolve conducting filament by repelling or absorbing oxygen vacancies to create LRS and HRS, respectively [21]. (b) PCM device. When the pulse is stopped abruptly, the molten material quenches into the amorphous phase due to glass transition. When a current pulse of lesser amplitude is applied to the PCM device in the HRS state, a part of the amorphous region crystallizes. By fully crystallizing the phase change material, the LRS state is obtained [52]. (c) STT-MRAMs device. The device contained two ferromagnetic layers and one dielectric tunnel barrier in between. The polarization of the top layer (free layer) can be rotated during the programming phase by changing the polarity of the voltage (current). The device will be in LRS (HRS) if the two ferromagnetic polarizations are parallel (anti-parallel) [53].

2.2.2. PHASE CHANGE MEMORY

Phase change memory (PCM), which also dates back to the 1960s35, is based on the property of certain types of materials, such as Ge2Sb2Te5, to undergo a Joule heating-induced, rapid and reversible transition from a highly resistive amorphous phase to a highly conductive crystalline phase. A typical PCM device has a mushroom shape where the bottom electrode confines heat and current. This results in a near-hemispherical shape of the amorphous region in the HRS state. By crystallizing the amorphous region, the LRS state is obtained [52]. The device in LRS and HRS states is illustrated in Figure 2.3(b).

2.2.3. SPIN TRANSFER TORQUE MRAM

Magnetic tunnel junction (MTJ) is the data-recording element in Spin transfer torque MRAM (STT-MRAMs); it encodes two bi-stable magnetic states into one-bit data. Fundamentally, the MTJ consists of two ferromagnetic layers sandwiching an ultra-thin dielectric tunnel barrier (TB). The top ferromagnetic layer is named as free layer (FL), where the magnetization can be switched by applying a spin-polarized current going through it. The bottom ferromagnetic layer is called pinned layer (PL), where the magnetization is strongly pinned to a certain direction. Therefore, the FL's magnetization can be either parallel (P state) or anti-parallel (AP state) to the PL's [53]. The state of the device in LRS

and HRS is illustrated in Figure 2.3(c).

2.2.4. OVERVIEW OF MEMORY TECHNOLOGIES

Regardless of the memory technology, there are some essential characteristics that we always need to consider when it comes to memory. Access time is how fast information can be stored and retrieved (read and write latency). Cycling endurance is the number of times a memory device can be programmed (or switched). Finally, the access energy is the energy consumed to read from or write into the device. Table 2.1 provides a rough comparison between mainstream memories and emerging memories with respect to some important characteristics.

Table 2.1: Comparison of mainstream and emerging memories [6, 7, 11]

	Mainstream Memories			Emerging Memories		
	SRAM	DRAM	Flash	RRAM	STT-MRAM	PCM
Size (F^2)	120-150	10-30	10-30	10-30	10-30	10-30
Volatality	Yes	Yes	No	No	No	No
Voltage	< 1 V	< 1 V	>10 V	< 3 V	< 1.5 V	< 3 V
Write energy	~fJ	~10 fJ	~100 pJ	~1 pJ	~1 pJ	~10 pJ
Write latency	~1 ns	~10 ns	~1 ms	~10 ns	~5 ns	~10 ns
Read latency	~1 ns	~3 ns	~100 ns	~10 ns	~5 ns	~10 ns
Endurance	~10^16	~10^16	~10^4 - 10^6	~10^7	~10^15	~10^12
Scalability	Medium	Medium	Medium	High	High	High

In Table 2.1, F denotes the minimum feature size. According to the table, we can see emerging memories provide high density. If emerging memories can also support multi-bit storage, their density would be even higher than mainstream memoirs. Next, they are non-volatile despite SRAM and DRAM. This leads to lower static energy consumption. They also operate on lower voltage levels than flash memory which makes them good candidates for embedded applications [11]. The read and write time is worse than SRAM, comparable to DRAM, and significantly better than Flash. The endurance of emerging memories is a bit behind volatile memories, but comparable and better than Flash memories. Finally, these emerging memories often suffer from high programming energy. This is the main reason that researchers often use these devices for applications that do not require frequent reprogramming.

Despite the promising characteristics of memristors, their non-idealities are a significant concern in the context of reliable computing. These non-ideal behaviors are mainly due to device imperfection fabrication and inaccurate programming. In the following, we briefly elaborate more on different types of non-idealities based on the information provided in [9, 54].

- Programming Noise: In an ideal case, memristors can alternate between certain conductance levels with unique values. However, in reality, each level can have a range of conductance values with a Gaussian distribution. The variation in the programming voltage (or current) is highly correlated with the programmed value. For the memristors with high conductance levels, the noise margin between the levels is re-

duced. Hence, the adverse impact of programming noise is more detrimental. Therefore, using a large-resolution memristor cell is challenging for a system in terms of accuracy/functional correctness.

- IR-drop: IR-drop is caused by wire resistance. As the memristor gets far from its driver, the impact of wire resistance increases. Therefore, the voltage required for programming and reading is dropped from its nominal value. As a result, the memristor is programmed into the wrong value (programming noise), or a wrong value is read from the device during a read operation.

- Non-zero G_{min} error: The information is represented by different resistance levels in the memristors. As an example, logic value '0' should be mapped to a high or low resistance level. In both cases, there would be a current where it goes through to the sensing circuit. In other words, a non-zero output current is produced when a non-zero input voltage is applied to a memristor with a high resistance level representing digital zero. This can be challenging when we want to perform mathematical operations with memristor devices [55].

- Endurance and Retention: The endurance (aging) problem comes from the destructive nature of the programming operation. For example, in the case of ReRAM device, this aging might be caused by a change in the oxygen vacancies due to oxygen diffusion. The endurance issue sets a limit on the number of times we can reprogram these devices. This is why researchers often consider memristors for applications that do not require frequent reprogramming. Currently, the endurance of memristor devices is above 10^6. The retention time for memristor devices can also go up to 10 years [9].

- Conductance drift: The conductance of memristor devices drift during the time. This happens by even reading the device frequently. Hence, after a certain period of time, the device has to be reprogrammed to return to its original state.

- Read disturb: Read disturb is a severe conductance drift where the actual value is flipped after a read operation.

2.3. MEMRISTIVE CROSSBAR STRUCTURE

By placing the memristor devices in a crossbar structure, not only they can be used as a storage unit, but some functionalities can be enabled in the memory as well. There have been proposed different crossbar structures. In the following, we describe the most common structures briefly.

2.3.1. 1R CROSSBAR ARRAY

Figure 2.4(a) depicts a memristive crossbar array where each cell comprises a single memristive device (1R). The cells are accessed by two lines, word-line (WL) and bit-line (BL). This type of array structure provides high density compared to other array structures. In order to read a device, we need to drive the desired WL with 'V_{read}' while the rest are connected to the ground. Then, the value of all the memristors, placed in the active row, is read by sensing the current flows in the bit-lines. However, the drawback

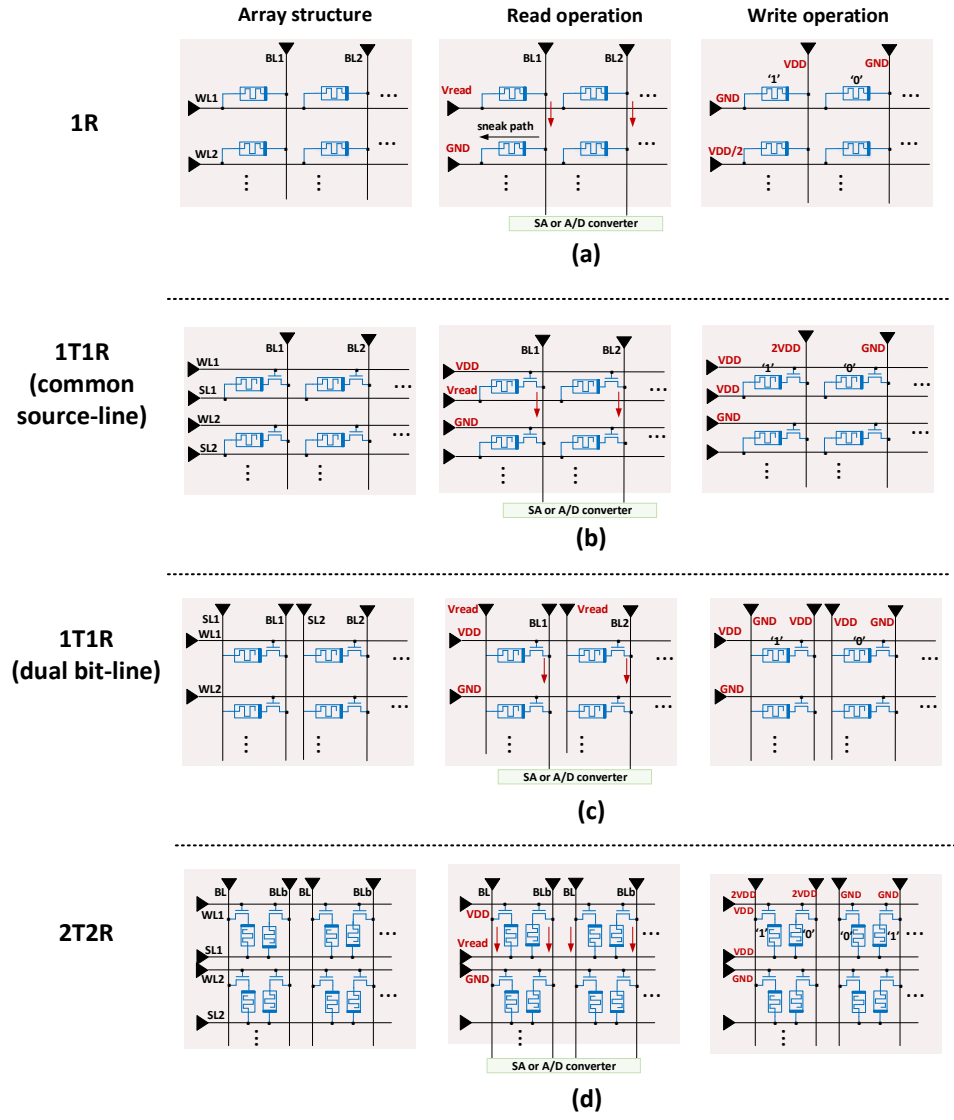


Figure 2.4: Common crossbar structures and how they operate during read and write operations. a) Passive crossbar with high density but suffering from sneak path current. b) 1T1R common source-line array tailored for VMM. C) 1T1R dual bit-line array suitable for streaming logical operations. d) 2T2R crossbar structure achieves high performance and sensing margin.

of this design is its sneak path current. As the figure illustrates, the bit-lines current may

leak via other memristors in the inactive rows. Hence, the current sensed by a sense amplifier may not correspond to the value stored in the memristors.

To program the devices, we must perform two steps: 1) initialization and 2) writing. In the initialization step, the devices of a selected row are reset to HRS (logic '0') by applying VDD to the WL and GND to all the BLs. In the writing step, GND is applied to the selected WL, and VDD to only the BLs where we want to program their memristor to LRS (logic '1'). We need to apply $1/2(VDD)$ to non-selected WLs. This prevents non-selected devices from switching. According to the figure, the voltage across the selected device in the first column is VDD (i.e., $VDD > V_{set}$), while $VDD - VDD/2$ across the deselected device in the second column (i.e., $VDD - VDD/2 < V_{set}$).

2.3.2. 1T1R COMMON SOURCE-LINE CROSSBAR ARRAY

To eliminate the sneak path current, we can use an access transistor next to each memristor device. Figure 2.4(b) depicts 1T1R crossbar structure where the memristors in a row share a common source-line. To read the value of a memristor, we apply VDD on its corresponding word-line to turn on the access transistor. Besides, V_{read} is applied to the source-line and the current (or voltage) is sensed on the bit-line. To program all the memristors placed in one row in a single cycle, we need two different voltage levels; VDD and $2VDD$. We drive the source-line with VDD, and depending on whether we want to set the device to LRS (logic '1') or HRS (logic '0'), $2VDD$ or GND is applied on the bit-lines, respectively. This structure is suitable for the vector-matrix multiplication operation. We will explain this in the following sections.

2.3.3. 1T1R DUAL BIT-LINE CROSSBAR ARRAY

This structure also considers one access transistor for each memristor device. However, the source-line is shared among the memristors placed in one column of the crossbar. Figure 2.4(c) shows this structure. Dual bit-line arrays have lower latency and energy. This is due to the lower voltages during the write operation, meaning thinner oxide access transistor, and hence lower transistor effective resistance. On the contrary, this array structure needs more area compared to the common source-line array due to the parallel orientation of BLs and SLs. This structure is suitable for 'streaming' logical operations. We will explain this in the following sections.

2.3.4. 2T2R CROSSBAR ARRAY

Figure 2.4(d) shows the structure of the 2T2R crossbar where each cell comprises two memristors (2R) and two access transistors (2T). For each cell, the actual data and its complementary value is stored. This structure requires a differential sensing mechanism (similar to traditional SRAM). The data and its complementary value go through the bit-line and bit-line bars. A slight difference between the voltage or current of these two lines is sensed using a differential sense amplifier. This leads to better performance, higher sensing margin, and more tolerance against variation and non-idealities. This is achieved at the cost of lower area efficiency. Besides, the efficiency of this structure for computational operations (e.g., VMM, XOR) has to be further studied. More discussion on this structure can be found in [12, 56].

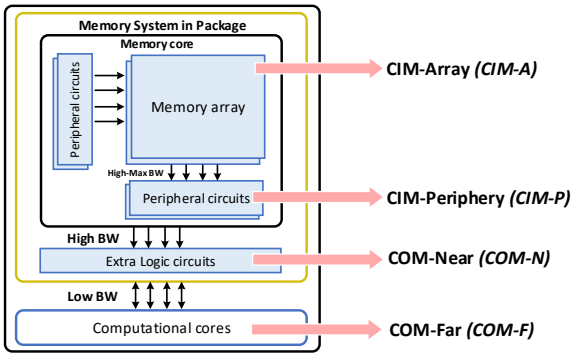


Figure 2.5: Illustration of four possible places to preform computation.

2.4. CLASSIFICATION OF MEMRISTIVE CIRCUITS

The architecture of computers consist of (one or more) memories and (one or more) computational units. A memory core consists of one or more cell arrays and peripheral circuits. Traditionally, the computing takes place in the computational cores. However, the notion of CIM aims to enable computation inside the memory. As we perform the computation closer to the memory, we may acquire higher bandwidth. Figure 2.5 indicates the four possibilities where a computation result can be produced. According to the classification, if the result of computation is produced outside the memory core, then it is referred to as Computation-Outside-Memory (COM). In this case, the computation can take place either in extra logic circuit inside the memory System-in-Packages (SiP) or like a traditional architecture in computational cores (CPU or GPU), which are referred to COM-Near (COM-N) or COM-Far (COM-F), respectively. An example of COM-N is 3D-Stacked Memory where extra logic is placed in memory system to perform some computation. The two main classes of CIM circuit designs using memristive devices where the results produced inside the memory core are described in the following:

CIM-Array (CIM-A): The result of the computation is produced within the array and represented by resistances. CIM-A circuit designs have generally the following advantages in common:

- They achieve the maximum bandwidth of "transferring" data between the computation and storage units, as both computation and storage happen physically within the same memory array.
- They perform computation independently from the sense amplifiers. This can provide high parallelism.
- They can potentially enable logic cascading of universal functions.

However, they share some limitations as well such as:

- Frequent write operations can lead to endurance and energy issues.

- Performance overhead due to the high latency of device programming and logic cascading to achieve complex functions.
- Significant design efforts both for the memory array (e.g., the memory array must support wider bit-lines to handle the excess of currents when multiple values are written simultaneously) and its controller (e.g., the controller requires complex state machines to apply a long sequence of different control voltages to the memory array).

CIM-Periphery (CIM-P): The result of the computation is produced within the periphery and represented by voltage levels. CIM-P circuit designs generally have the following advantages in common:

- They do not exacerbate the endurance of the memory array as the memory states do not change during and after the computation.
- They require relatively less redesign efforts in the memory array; i.e., the currents during computation are lower or close to the write current.
- They allow high to maximum bandwidth depending on the complexity and area of the periphery.

However, they share some limitations such as:

- They may have to share sense amplifiers or analog-to-digital converters per multiple bit-lines, which can degrade the performance.
- They cannot cascade functions without read/write operations.

In the following, we dive into the detail of a CIM-tile (memory core) to understand how the data flow in a tile and what the main steps are. Based on that, we collect and classify main operations we can preform in a CIM-tile. Then, we provide a brief explanation about the implementation of each operation.

2.4.1. GENERIC ILLUSTRATION OF CIM-TILE

In an attempt to potentially define a generic CIM-tile design, one should first establish a better understanding of all potential capabilities of memristor-based circuits as well as a deep understanding of the data flows within a CIM-tile for a wide range of applications. Hence, we first present a logical breakdown of the data flow within an abstract CIM-tile and its potential to create complex functions. Second, we investigate the state-of-the-art memristor CIM-based logic and arithmetic functions.

A CIM-tile comprises a memristor crossbar and its digital as well as analog peripheries. Figure 2.6 depicts a generic CIM-tile where the tile receives digital data as well as instructions [14] (or control signals) as inputs. The controller inside the tile is responsible to orchestrate all the circuits according to the instructions. The data may pass through four steps, 1) Input processing f_{in} 2) Crossbar array f_{Xbar} 3) Sensing f_{sens} 4) Output processing f_{out}. Until now, some novel circuit designs were proposed to expand the number of functionalities that can be supported in each of these steps. Based on this knowledge, when an application is intended to be executed using CIM-tiles, the designer

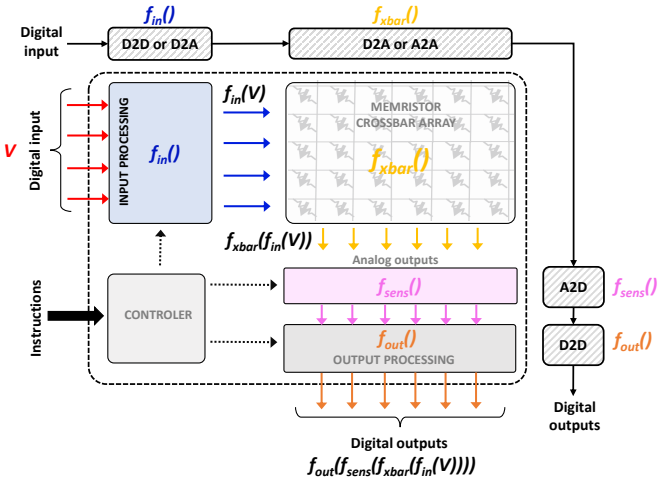


Figure 2.6: Generic illustration of CIM-tile comprising four steps: 1) input processing 2) crossbar array 3) sensing 4) output processing.

selects a certain function and its associate circuit for each of those steps. Hence, we will end up with an accelerator designed just for a certain application. Until now, the main focus of researchers regarding memristor-based CIM was on designing accelerators that can only benefit specific applications. However, in order to have an as general as possible design, a wide range of functionalities should be supported in each of these steps at the same time, rather than having a different circuit for each function. In the following, we will describe what primitive functionalities are already supported in these four steps. Identifying these functions also enables us to build more complex functionalities in the CIM-tile.

2.4.2. PRIMITIVE FUNCTIONS IN THE CROSSBAR - f_{Xbar} (CIM-A)

1. **NOR** [57]: As illustrated in Figure 2.7, in order to perform the *NOR* function, two memristors are used to represent the two input operands and the result is written to a third memristor. At first, the output memristor should be initialized to Low Resistance State (LRS) representing a logic 1. Afterward, voltage level *V0* is connected to the bit-lines of the input memristors while the output memristor is connected to the ground. When one or two inputs are a logic 1, the voltage level *V0* should be sufficient enough to be able to change the resistance of the output memristor without switching the resistance of input memristors.

2. **NoT** [57, 58]: Figure 2.8 shows two different implementations for *NOT* based on (a) Sinder [58] and (b) MGIC [57]. Considering Snider implementation (Figure 2.8 a), one device holds the input, while the output is produced in the second device. The load resistor R_g is an integral part of the design, and it is constrained to $R_{on} \ll R_g \ll R_{off}$. The required control voltages V_{wh} and V_{ω} are constrained to $0 < V_{wh} < V_{set} < V_{\omega}$,

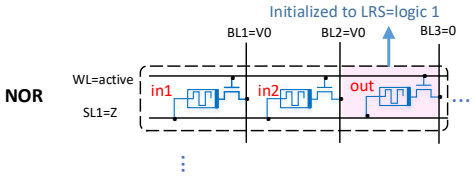


Figure 2.7: Implementation of a NOR function.

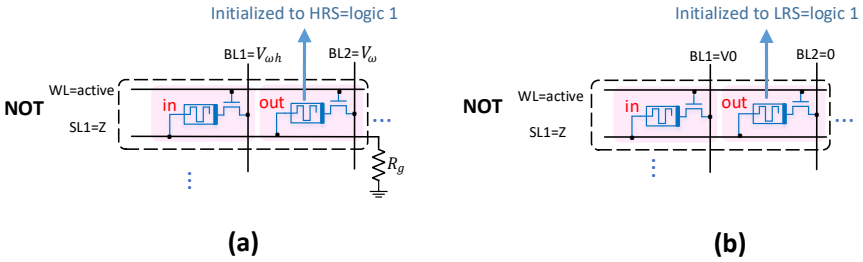


Figure 2.8: Implementation of a NOT function based on (a) Snider and (b) MAGIC.

$V_{\omega} - V_{wh} < V_{set}$ and $V_{reset} < V_{\omega}$. The execution of the *NOT* function requires two steps. In the first step, the output device is initialized to R_{off}. In the second step, V_{wh} and V_{ω} are applied to the bit-lines according to the figure, while the source-line (SL) is left floating. When the input is 1 (R_{off}), a zero voltage is produced on the horizontal line leading to a voltage V_{ω} across the output device. Therefore, the resistance state of the output device switches. When the input is 0 (R_{on}), V_{wh} is produced on the horizontal line. Hence, a voltage of $V_{\omega} - V_{wh}$ is produced across the output device, which is insufficient to switch its resistance state [59]. Figure 2.8(b) shows the MAGIC implementation. This is the same as the NOR implementation mentioned before. By programming one input of the NOR gate to logic 1, the NOT function can also be implemented.

3. **OR** [60, 61]: The schematic of an OR function depicted in Figure 2.9(a) is similar to the NOR function. However, the output memristor is initialized to the High resistance State (HRS). In addition, despite the NOR operation, the bit-lines of input memristors are grounded while the bit-line of the output memristor is connected to voltage V0. Similar to a NOR, V0 should be determined to fulfill this functionality without being destructive. Despite the first implementation, the second implementation (Figure 2.9(b)) receives the inputs both as a voltage and resistance level. The output is overwritten into the memristor (destructive function). If the memristor has been initialized to LRS, (here is logic 1) or V_{ω} applies to the bit-line, the final state of the memristor will be LRS (logic 1).

4. **Minority** [60] Considering the NOR function, having one of the input memristors as a logic '1' is enough to switch the resistance of the output memristor. Accordingly, by

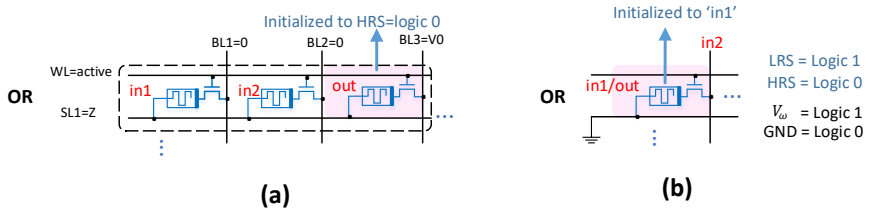


Figure 2.9: Two different implementations of OR function.

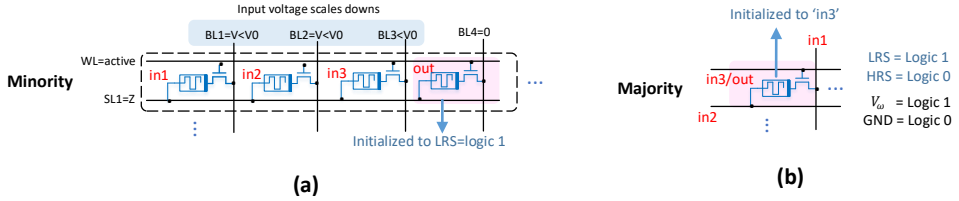


Figure 2.10: (a) Minority and (b) Majority implementation.

reducing the voltage of V0, we can determine the minimum number of ON resistances to be able to switch the output device. This can effectively implement the minority function. Figure 2.10(a) illustrates the implementation of this design.

5. **Majority** [61]: This function has three inputs, IN1, IN2, and IN3, and one output, OUT. The states of IN1 and IN2 are voltage, while the states of IN3 and OUT are resistive; i.e., logic '0' and '1' are represented by R_{off} and R_{on}, respectively. The execution of the Majority function requires a single step in which the inputs IN1 and IN2 are applied to the bit-line and source-line. In addition, the memristor should be programmed to IN3. Depending on the inputs, the memristor device may or may not switch. The circuit implements the $Majority(IN1, \overline{IN2}, IN3)$ [59].

6. **AND** [61]: The circuit of the AND function presented in Figure 2.11 is similar to the Majority function explained before. Here, the select-line is connected to V_ω. The two input operands are presented by the voltage applied to the bit-line and the resistance level of the memristor. The result is programmed into the memristor. In order to get logic '1' (LRS) in the output, the device should be initialized to LRS, and V_ω should be applied to the bit-line.

7. **NAND** [58, 60]: Figure 2.12 shows two implementations for the NAND function. The execution for both implementations requires the initialization of the output device. For the first implementation (Figure 2.12(a)), the output device is initialized to R_{off} representing logic '1'. In the second step, V_{wh} is applied to bit-lines of the first and second inputs, V_ω is applied to the bit-line of output, while the source-line is left floating. When both inputs are 1 (R_{off}), a zero voltage is produced on the horizontal line.

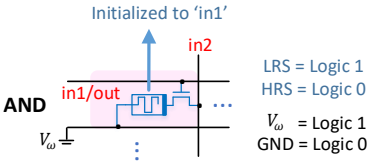


Figure 2.11: Memristor-based AND function circuit.

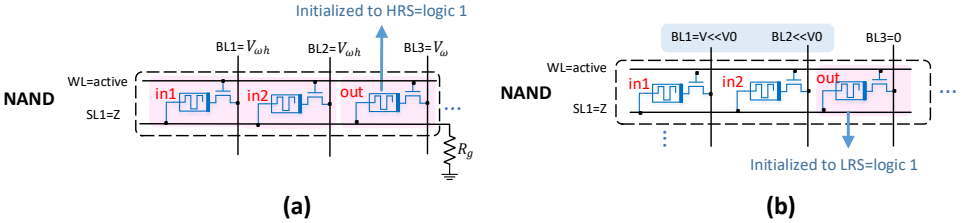


Figure 2.12: Two different designs for the NAND function.

This produces a voltage V_w across the output device, switching its resistance from R_{off} to R_{on}. On the other hand, when at least one of the inputs is 0 (Ron), V_{wh} is produced on the horizontal line. Hence, a voltage difference of $V_\omega - V_{wh}$ is produced across the output device, which is not sufficient to switch its resistance state. The second implementation (Figure 2.12(b)) is based on the Minority implementation explained before. By further reducing the voltage V0 for the Minority implementation, at one point, all the memristor inputs should be R_{ON} to be able to switch the output memristor, which can implement the NAND function.

8. **XOR** [62]: This function requires five memristor devices, among which two of them are auxiliary memristors. Figure 2.13 depicts how these memristors should be connected and initialized. In the case memristor devices, representing input operands, are both either ON or OFF, the common node between memristor devices is virtually ground, keeping the output memristor at its original state. Otherwise, the voltage on the common node is Vx which switches the output memristor to the LRS state with the help of auxiliary memristors. The crossbar structure and the way data is mapped to it to support this function is also demonstrated in Figure 2.13. More information can be found in [62].

9. **Copy** [58]: Figure 2.14(a) shows the schematic of the Snider Copy function. As usual, in the first step, the output device is initialized to R_{off}. Then, GND and V_ω are applied to the bit-lines of input and output devices, while the source-line is left floating. When the input is 1 (R_{off}), the voltage divider between the devices ensures voltage drops of ($V_\omega/2$), which is not sufficient to switch their resistance states. However, when the input is 0 (R_{on}), almost V_ω is applied across the output device. This is sufficient to

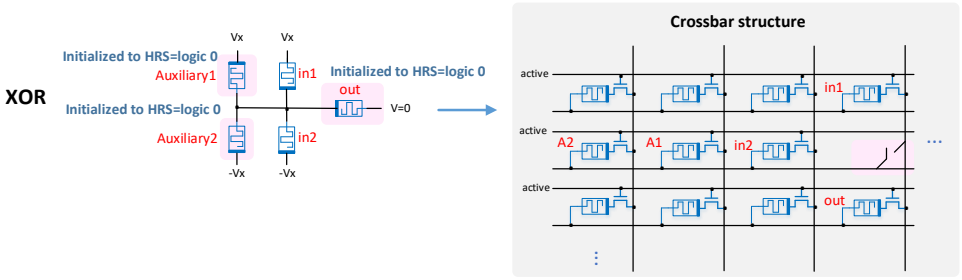


Figure 2.13: Schematic of XOR function and its required crossbar structure.

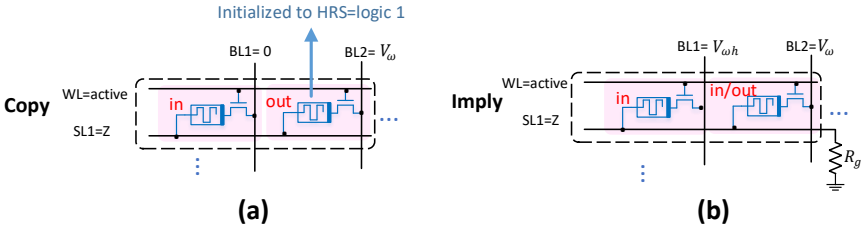


Figure 2.14: Implementation of (a) Snider Copy and (b) Imply functions.

switch the device from R_{off} to R_{on}.

10. **Imply** [59]: The material implication is a logic function that implements conditional statements ($p \rightarrow q \Longleftrightarrow \sim p \vee q$). Figure 2.14(b) presents the implementation of this function. In the first step, the memristors are initialized with the input values. Then, according to the figure, proper voltages are applied to the bit-lines. When the output device (q) is 1 (Ron), there is no way to switch it to R_{off}. The only way to obtain R_{off}, as a result, is to initialize the output to R_{off} and the input to R_{on}. This implements material implication.

By cascading the aforementioned functions inside the crossbar, more complex functions such as arithmetic addition or multiplication, can be implemented. However, as demonstrated before, different functions require different voltages and crossbar structures. This becomes even more challenging when the sensing step and its functions are taken into consideration. Therefore, finding a unique solution that can cover most of these functions with minimum cost is a valuable step toward a generalized memristor-based CIM-tile. In addition, having a smart tile controller to deal flexibly with different execution flows and patterns of data mapping is an essential part of this system.

2.4.3. PRIMITIVE FUNCTIONS IN THE SENSING STEP f_{sens} (CIM-P)

Despite the functions mentioned before, here, the results of the functions are generated by using sense amplifiers (SA) or analog-to-digital converters (ADC). In other words, the

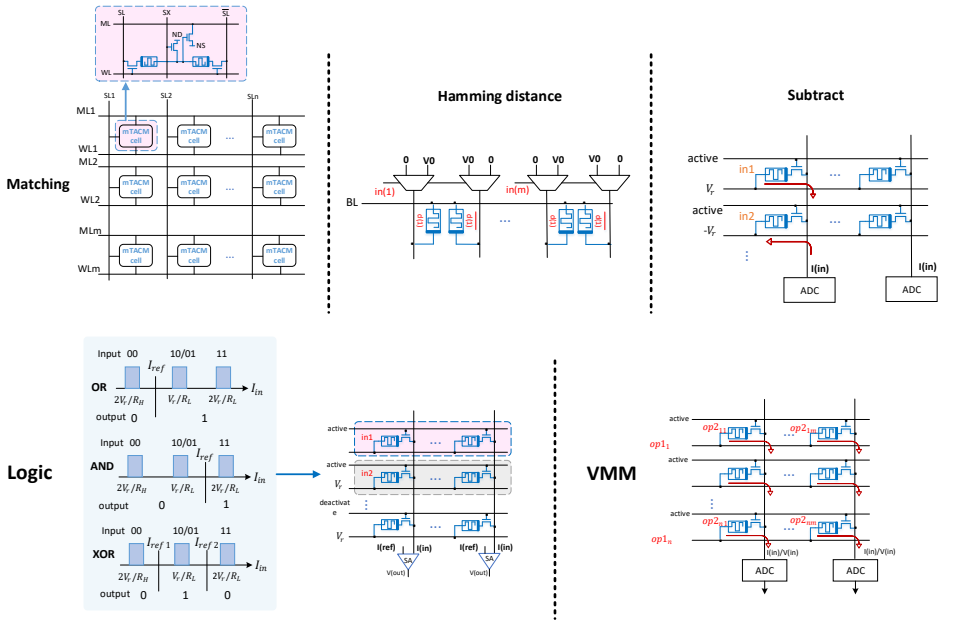


Figure 2.15: Primitive functions supported inside the sensing step (f_{sens}). This figure presents five operations (Matching, Hamming distance, Subtract, Logic, and VMM) that can be supported in an analog way using memristor crossbars. The result of these operations is produced in the sensing stage.

current generated by the crossbar is assigned to different logical values, which in turn can implement different functions.

1. **Hamming Distance [63] and Matching [64]**: Using a CIM-tile, the Hamming Distance function can be implemented. To calculate the distance between two vectors, one vector, and its complementary value have to be programmed to the crossbar. As depicted in Figure 2.15, the second vector is given to the select signals of the pair of multiplexers placed on top of the crossbar. This provides data as well as its complementary value to the memristors holding the first vector and its complement, respectively. Subsequently, based on the similarity of the two vectors, different current levels flow into the bit-line, which is sensed to determine the distance of the two vectors. Similarly, a Match function can be performed by just using a SA. A potential crossbar structure for this function [64] is depicted in Figure 2.15.

2. **Subtraction [65]**: Element-wise subtraction can be performed in the crossbar. The proper voltage level has to be provided to the row corresponding to the negative operand to be able to drain the current from the bit-line. Hence, the remaining current on the bit-line represents the result of this function.

3. **Scouting Logic (AND-OR-XOR) [66]**: Logical operations can be implemented in the sensing step. In this approach, both operands as vectors are programmed to the crossbar. As illustrated in Figure 2.15, depending on where to put a reference(s) of SA, logical OR, AND, and XOR functions can be implemented. The main advantage of this approach compared to implementing these functions inside the crossbar is reducing the number of device programming. This is important due to the endurance problem and high programming energy of memristor devices.

4. **Vector-Matrix Multiplication**[67]: Vector-Matrix Multiplication is the main functionality for which CIM-tile is intended to be employed. To perform this function, first, the matrix has to be programmed to the crossbar. Subsequently, the vector is applied to the select line of the crossbar. Finally, all the elements of the output vector are obtained in one shot after converting analog current/voltage levels to their digital value using ADCs.

Based on the example of functions in the sensing step, when a designer intends to perform an application using one of those functions, having the competence to do other functions becomes challenging due to the necessity of different array structures, circuits, and constraints. As an example, Scouting logic just requires SA with two determined references with a constraint on activating two rows at a time. However, in order to generalize the design for other functions like VMM, ADCs have to be placed, and more rows potentially should be activated.

2.4.4. PREVALENT FUNCTIONS IN THE DIGITAL PERIPHERY f_{out} (CIM-P)

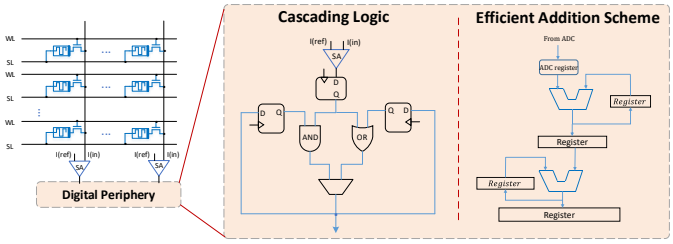


Figure 2.16: Examples of digital periphery circuits designed to deliver certain functionality (f_{out}).

Due to the complexity of the application execution flow and the limitation of functionalities in CIM, some parts of the application should be performed by the host. However, researchers try to place customized circuits in digital peripheries to expand the functions that can be done in the memory and gain more energy/performance improvement rather than simply assuming unsupported parts of an application are assigned to the host processor.

Many circuits have been considered in the digital periphery for different applications. Figure 4.5 shows two examples: 1) Cascading logic circuit [68] is customized digital periphery to perform sequential AND and OR logic in the periphery targeting database ap-

plications. 2) Efficient Addition Scheme [26] is a novel structure to replace conventional *shift and add unit* targeting matrix-matrix multiplication kernel. Based on the applications, some designers also placed subtractor, sigmoid function, vector processing unit, and register files in the digital periphery. Due to the flexibility that can be provided by digital design compared to the crossbar and sensing step, more attention should be given to this step for a generalized CIM-based design. One example of periphery design that can support a wide range of functionalities is employing Lookup Table (LUT) [65, 69]. However, still more research has to be conducted to have a smart and general solution with a detailed execution model for a wide range of applications.

2.4.5. EXAMPLES OF FUNCTION IN THE INPUT PROCESSING STEP f_{in} (CIM-P)

Despite other steps, this step has not been investigated much yet in the literature. However, some digital circuits were designed in [14] to carry out some functions. First, a buffer using parallel-in-serial-out registers is considered to provide a clear separation of tile from outside and facilitate data communication considering datatype size and limited resolutions for Digital to Analog Converters (DAC). Second, a digital circuit is placed to be able to flexibly support different patterns of crossbar row selection. Other examples of this step could be Address CAlculation (ACA) [70], data quantization, downscaling, and alignment.

2.5. POTENTIAL APPLICATIONS

Table 2.2: List of some potential applications/algorithms/kernels that can be executed using memristor-based CIM.

Domain		**Applications/Algorithms/Kernels**				
Network		Automata processor for network security [71]	Packet classification [72]	SAMCRA [73]	IP-routing	-
	operation	AND	Matching	VMM	Matching	-
Security		Security primitives (PUF-RNG) [74]	Encrypted communication [75]	AES [76]	Regular expression matching [64]	Salsa20/ChaCha20
	operation	Analogue properties	Analogue properties	Add-XOR-Multiply	Matching	Addition - XOR
Approximate computing		Image property calculation [77]	Approximate matrix multiplication [78]	Quantum simulations [79]	-	-
	operation	Addition - Subtraction	VMM	VMM	-	-
Mathematics		Partial differential equation solver [80]	Eigenvector calculation [81]	Markov chain	Vector-cosine similarity	-
	operation	VMM	VMM	VMM	VMM	-
Signal and image processing		Compressed sensing [82]	Discrete Fourier Transform [83]	Convolutional Image filtering [84]	Huffman encoding	GLCM feature extraction
	operation	VMM	VMM	VMM	Matching	Addition
Classification and Prediction		Recurrent neural network [85]	Convolutional neural network [67]	Hyperdimensional computing [23]	Reservoir computing [86]	Auto-regressive models
	operation	VMM	VMM	AND - OR - VMM	Matching	VMM
Database		Query-06 of the TPC-H [87]	Pattern matching in databases	Transitive closure	Bitmap indices	BitWeaving
	operation	Bitwise XOR	Matching	VMM	OR - AND - XOR	OR - AND - XOR
Bioinformatics		DNA sequencing or allignment [88]	Genome Base-calling [89]	Genome Profiling [90]	-	-
	operation	XOR - VMM	VMM	VMM	-	-
Graph processing		Graph Clustering [63]	Breadth first search [91]	VLSI routing	PageRank	All-pair-shortest-path
	operation	VMM - NOR - HD	VMM	VMM	VMM	VMM

Besides the existing operations, a designer should have a good insight into the applications that have the potential to be executed using a memristor-based CIM system. Table 2.2 presents the list of some applications/algorithms/kernels where a considerable part

of the program can be performed inside the memristor-based memory and potentially gain performance and energy from the concept of CIM. The examples in the table cover a wide range of domains. This helps the designer to expand the generalization of the design by finding out the corner cases regarding application requirements and constraints. It should be noted that these applications were evaluated by academic papers. Hence, further evaluation with more accurate industrial models is required to ensure that CIM-based implementation for these applications can outperform the digital CMOS-based counterpart. In the following, some important aspects to be considered for a generic CIM system design targeting a wide range of applications or even accelerator design are listed:

1. Different applications require computation over different data types and data type sizes. A generalized design should enable computation for some basic **data types**. Besides, a design that can flexibly and efficiently support computation for different datatype **sizes** is valuable since it can potentially lead to energy/performance improvement. This is also true in the case of accelerator design. A designer should be aware of the required data types for the intended application(s) and enable them in the design.

2. As shown in Table 2.2, different applications consist of different **operations**. However, according to what was discussed before, a designer should realize a system to cover more functionalities. This results in offloading fewer parts of the program into the host and, in turn, less communication between the host and storage unit.

3. Based on the execution flow of applications, they may require different **patterns** on crossbar rows and columns selection. In other words, during the execution time, an application may operate on different locations of a crossbar. Therefore, this is important that a generalized system provides this flexibility and accessibility to a programmer.

4. Memristor crossbars have limited dimensions which are usually much less than what is required for an application. Considering that, the application should be mapped to several tiles. Hence, a detailed implementation of **tile communication**, which can flexibly cope with different data flows, is essential.

5. Variability is one of the challenging aspects of memristor devices, which can end in **accuracy** reduction. Some applications can tolerate this inaccuracy, while others need precise computation. Therefore, supporting algorithms to dynamically adjust the accuracy of computations based on the application requirements can be another research direction for a generalized system.

6. Although it is desired to offload more parts of an application on CIM-tiles and reduce data movement more, there might still be some parts that have to be executed on the host side. Accordingly, this indicates the necessity of **communication** and synchronization of CIM-tiles with the host. Based on the existing works in the literature, a detailed design regarding this aspect is missing and requires more attention from the community.

2.6. CIM ABSTRACT DESIGN CHOICES

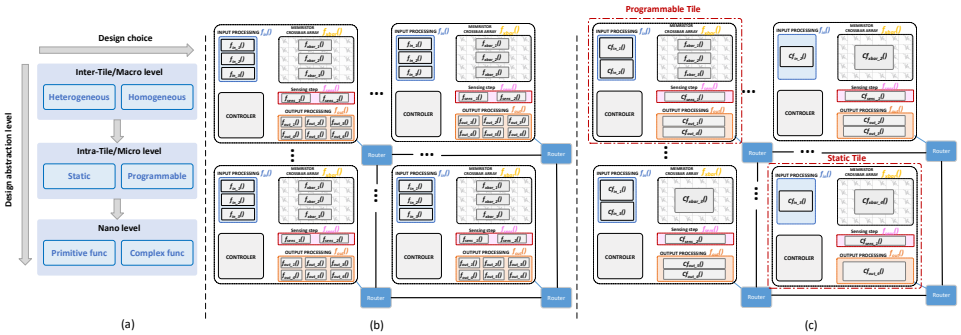


Figure 2.17: (a) Possible design choices at different abstraction levels for CIM. (b) Homogeneous design where each tile is capable and generalized enough to support a wide range of functionalities. (c) Heterogeneous design where tiles are customized to execute different kernels efficiently.

As we can perceive from previous sections, there are two general directions toward how to employ CIM. The common direction so far is designing accelerators for target applications. These accelerators are designed only for fixed data-flow and control-flow with minimal flexibility. Although a designer can heavily optimize the accelerator, this approach suffers from limited applicability. Any changes from the application level may require designing a new accelerator. On the other hand, the second direction is going toward more generic designs. In the following, we scrutinize this direction more. In this thesis, we follow both directions in parallel.

As discussed before, despite application-specific designs, in order to have a memristor-based CIM-tile system capable of executing a wide range of applications, more complexity and requirements are raised. To move toward such a system, different strategies can be applied with respect to different abstraction levels. As illustrated in Figure 2.17 (a), the system can be broken into at least three levels. Starting from the lowest level, Nano-level, a CIM-tile can comprise just a primitive function in one of the steps, while no major functionalities supported in other steps in the tile. An example of this can be MAGIC [57] or Scouting [66] where just primitive functions supported in the crossbar f_{Xbar} or sensing steps f_{sens}, respectively. On the other hand, different functionalities can be supported at different steps of the tile in order to end up with a complex function. In the second level of abstraction, Intra-Tile/Micro level, a Static or Programmable CIM-tile can be targeted. In the case of Static, each step of the tile is designed (and **C**ustomized) to provide specific functionality, while in the case of programmable tile, more than one function is considered for the tile's steps. Regarding the design choices in the highest abstraction level and considering a spectrum, on one side, a designer can aim at a homogeneous CIM-tiles system where the tiles are the same, and each can support a wide range of functionalities and requirements. On the other side of this spectrum, we will have a heterogeneous design where CIM-tiles are customized for different purposes, and

they can all provide this generality for the programmer. Finally, in the middle, we also may have a hybrid solution where limited heterogeneity is considered while CIM-tiles are pushed to support different functions as much as possible.

Figure 2.17 (b and c) illustrates the design choices for the inter-tile abstraction level. Considering a homogeneous design, in order to execute an application, more resources are available since the tiles are generalized and can cope with different functions. However, this might come at the cost of an area, energy, and latency overhead. In addition, due to the complexity of the tile, an advanced controller has to be employed. Instead, considering heterogeneous design, while fewer resources might be available for a certain application, the tiles might have higher energy, area, and performance efficiency. Nonetheless, due to the heterogeneity, a more complex task offloading scheme might require. Finally, the communication between tiles may impose more overhead when the tiles selected for an application are far from each other. The design choice heavily depends on how flexible the circuit in CIM tile will concern the applications' functionalities and requirements. No matter which approach is chosen, flexibly controlling this storage unit both in the granularity of inter- and intra-tiles for different execution flows is a crucial aspect. This can be achieved either all in hardware or partially at the software level with the help of a compiler (or scheduler) [21]. This becomes more important for heterogeneous designs when different workloads have to be mapped and scheduled for specific CIM-tiles. In conclusion, 1) expanding the functionality of a CIM-tile with a unified circuit, 2) flexibly interfacing the CIM-tiles to each other as well as the host, and 3) an advanced controlling system able to flexibly cope with different scenarios, are the key aspects toward a generalized CIM-tile design.

2.7. OVERVIEW OF STATE-OF-THE-ART DESIGNS

In this section, we briefly explain the main related works to this thesis. We categorize them into two parts according to what we have discussed before. First, we talk about the works presenting generic designs for CIM. Then, we go through three main promising applications and see the main accelerators designed for them.

2.7.1. GENERIC DESIGNS

There are very limited works doe far attempting to enable the concept of CIM for a generic platform. UPMEM [92] is the first commercialized generic platform for CIM. This platform is based on computation next to DRAM memory arrays rather than using emerging non-volatile memories. UPMEM consists of many data processors (DPU) uniquely positioned within the DRAM memory chips and next to the data, to compute data-intensive operations while drastically reducing off-chip data movements. Those DPUs are controlled by the high-level application running on the main CPU that ensures task orchestration. Similarly, Ambit [93] enables some primitive logical operations again inside DRAM memory. In the following, we briefly discuss the works aimed at using CIM in a generic direction using memristor devices.

- **PUMA**[16]: PUMA is the first design that brings general-purpose CIM with the help of programmability. The designed architecture has three pipelined stages (fetch, decode,

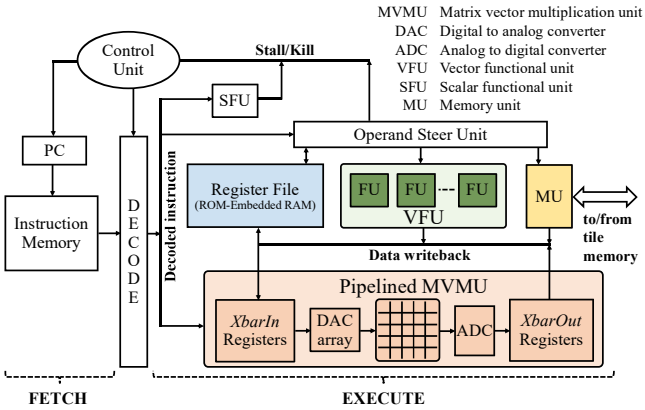


Figure 2.18: Core architecture of PUMA [16]. Each core comprises a crossbar and many peripheral circuits.

and execute). PUMA supplements crossbars with an instruction execution pipeline and a specialized ISA. The new in-memory instructions defined there are mainly responsible for communicating data between memory units or performing scalar operations in digital peripheries. Figure 2.18 shows the architecture of PUMA. In this design, many vector and scalar functional units are placed in the periphery to support variety of operations next to the crossbar. PUMA also has its own compiler, translating high-level code to PUMA ISA.

- **In-memory data parallel processor**[65]: In this promising work, new in-memory instructions were proposed. The instructions are complete operations that have to be performed on crossbars based on the Single Instruction, Multiple Data (SIMD) execution model to enjoy massive data parallelism. The basic operations supported in the crossbar are multiplication, addition, and subtraction. In this work, communication between the crossbars was illustrated behaviorally without detail. Shared buses and H-Tree networks are chosen to manage the communication.

- **ReVAMP**[94]: ReVAMP integrates the resistive memory in a pipelined processor in which the memory replaces both the cache and register file. A new instruction is introduced (*Apply*) to perform a computation operation (Majority) in the memory crossbar. However, as we discussed before, emerging memories are slower and age faster than SRAM. Hence, replacing the cache and the register file with memristors can potentially slow down the speed of the processors.

Simulators: Many factors have to be considered when an in-memory memristor-based architecture has to be designed since they can have a non-negligible effect on the performance, energy, area, or even accuracy of the system. Therefore, the necessity of optimization and design space exploration at reasonable simulation time derive the researchers to develop high-level simulators for this new type of architecture. There are a few simulators, among which NVSim [95] and NVMain [96] are the first memory-oriented

simulators designed for non-volatile memories. These tools, which are inspired by CACTI [97], can estimate the access time, energy, and area of non-volatile memories. Similar to NVSim, MNSIM [18] proposed a behavioral simulation platform for neuromorphic accelerators that estimates design parameters using analytical equations. Besides, another system-level simulator [98] was proposed, in which by using probability functions to capture the accuracy of ReRAM cells, the behavior of an application in terms of accuracy can be evaluated. However, still there is a need for cycle-accurate simulators that actually execute in-memory instructions/operations. This allows the user to track all the control signals and the content of crossbar/registers and produces more accurate performance and energy consumption results.

2.7.2. APPLICATION-SPECIFIC DESIGNS

NEURAL NETWORK

Machine learning algorithms and specifically neural networks have pushed the state-of-the-art designs and become prominent in a variety of applications, including, but not limited to, language processing [99], object recognition [100], and image classification [101, 102]. Designing larger networks and the ability to train them with advanced algorithms was the main driver to enable performing complex applications for several years. Besides advanced algorithms, hardware implementation and its challenges play a major role in the deployment and development of DNN applications specifically for embedded systems. One of the main hardware challenges of NN is the considerable amount of parameters (BERT has around 110 million parameters) that has to be stored and performed computation on (the memory wall). The key computation required in neural networks is Matrix-Matrix multiplication. Besides, neural networks can typically provide a high degree of parallelism. All these reasons make the applications of neural networks the most popular use cases for CIM-based designs.

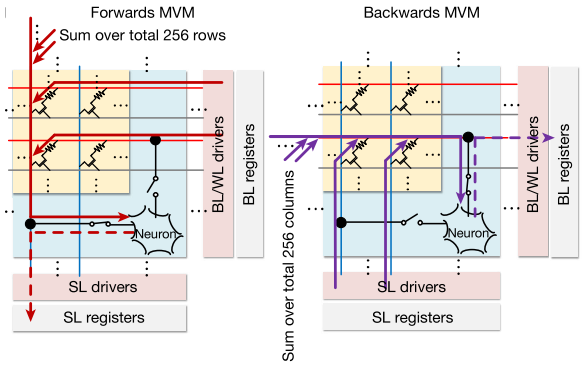


Figure 2.19: Implementation of bi-direction VMM [103].

There are several small-scale fabrications of accelerators for neural network applications. Authors in [103] fabricate 48 CIM cores, each containing 256 × 256 1T1R RRAM

cells and 256 CMOS neuron circuits, that can perform inference in parallel and supports various weight-mapping strategies. As illustrated in Figure 2.19, they provide bi-direction VMM by grouping 16 × 16 RRAM and sharing a neuron circuit among them. This might benefit different dataflow patterns at the cost of a complex crossbar structure and routing. Another work [104] reports the fabrication of high-performance and uniform memristor crossbar arrays for the implementation of CNNs. Each memristor array has an assembly of 128 × 16 1T1R cells. The design has eight arrays to implement a CNN with five layers and performs MNIST image recognition with 96% accuracy. HERMES [105] present 256 × 256 in-memory compute core designed and fabricated in 14-nm CMOS technology and multi-level phase change memory (PCM). The core also contains a current-controlled oscillator ADC and a local digital processing unit. More information about existing chips in this domain is provided in [106].

In the following, we classify accelerators based on network structure and highlight a few works per category. In most designs, the training is performed offline, and the accelerator is employed only for inference. However, a few works perform the training phase within the accelerator as well [107, 108]. Frequent device programming is required during the training. In addition, weight updates often require small changes in each iteration. Hence, more complex datatype and data mapping is required to be able to capture those small changes in the weights. Besides, more complex operations and data flow should be supported in the accelerators [109]. According to the above reasons, researchers often prefer to use CIM-based accelerators only for inference.

- Convolutional and deep neural network: PRIME [110] provides microarchitecture to accelerate neural networks using ReRAM devices. In PRIME, a portion of ReRAM crossbar arrays can be configured as accelerators for neural networks, while the rest is used as storage. ISAAC [67] proposes a pipelining mechanism for CIM-based implementation of convolutional neural networks to reduce the required buffer size between layers. Two accelerators [69, 111] designed intended for both training and inference. By analyzing data dependency, Pipelayer [69] exploits inter- and intra-layer parallelism. The authors in [111] analyze the variations resulting from the inherent characteristics of memristors and the errors of programming voltages on different networks.

- Spiking neural network (SNN): Memristors may be used for various devices in neural networks, i.e., neurons and synapses as well as neuronal circuits. Many works [112–119] in this domain aim to use memristors to model neurons. Traditionally, many transistors are needed when fabricating neurons by CMOS technology only. Hence, the goal is to use memristors to simplify the circuit. Authors in [120] harness the stochasticity of memristor devices to emulate the functionality of a spiking neuron. Authors in [121] propose a novel SNN using memristor-based inhibitory synapses to reduce the implementation complexity for lateral inhibition and homeostasis mechanism. There are also some works targeting memristor-based learning, mainly for supervised and unsupervised Spike Time Dependent Plasticity (STDP) learning [122–126]. More information about memristor-based SNN can be found in [127–129].

- Recurrent neural network (RNN): Recurrent neural networks are widely used for language modeling, speech recognition, translation, and sequence classification. The

structure and data flow for RNNs differ from other models since the current output depends on not only the input but also the previous output. Several works attempt to implement RNNs on memristor crossbars [85, 130–134]. Authors in [131] demonstrate an experimental implementation. This work shows that LSTM networks using memristor crossbar arrays can achieve high speed–energy efficiency. Another fabrication based on PCM devices is presented in [85]. This works demonstrate strategies for software weight-mapping and programming of hardware analog conductances that provide accurate weight programming despite significant device variability. More information on existing designs for memristor-based RNN is provided in [135].

- Binary neural network (BNN): Binary Neural networks, where the weights and activation values are binarized (-1,+1), receive more attention from researchers due to their high model compression rate and simplified computations [136]. This is appealing for edge devices where there are hard constraints on memory capacity, computing resources, and energy budget. Several designs propose different data representations and weight mappings for BNN [137–141]. In addition, an accelerator is designed in which memristors are also used as an activation function [142]. To ensure the accuracy of XNOR operations against device variation, a new memristor crossbar structure based on differential sensing is proposed [12]. More information can be found in [136].

GRAPH PROCESSING

Graph processing is employed in a wide range of areas, including but not limited to social media analysis [143], bioinformatics [144], urban planning [145], and machine learning [146]. Graph processing is well-known for its three main characteristics [91, 147]: 1) poor locality or random access pattern to the memory, 2) simple and a small amount of computation over the accessed data, 3) high sparsity of the graphs which implies that the computations are frequently performed on zeros operands. Traditionally, the active portions of the graph are loaded sequentially into the memory hierarchy of the system while the rest of the graph is stored in the secondary memory. Due to the explosion of graphs' size, data movement across the memory hierarchy imposes a considerable overhead compared to the actual computation time/energy and limits the system's performance due to the maximum memory bandwidth. The behaviors and characteristics of graph processing applications make them promising candidates to be accelerated by exploiting the concept of CIM.

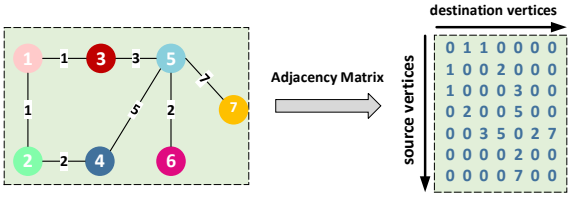


Figure 2.20: Representation of a graph using Adjacency matrix.

A widely used method of graph representation is an algebraic representation. In this

method, an *adjacency matrix* is constructed where each entry (i, j) represents an edge from source vertex i to destination vertex j. Figure 2.20 depicts the adjacency matrix of a simple graph. This representation allows for intuitive calculations using linear algebra operations. However, naive storage of this matrix in memory does not scale. The storage occupied by a 2D matrix grows quadratically, meaning that large graphs quickly occupy an impractical amount of memory. Usually, graph algorithms go through many iterations where a considerable amount of graph information stored in the memory has to be moved back and forth between the processor and memory. This imposes a significant overhead on the system, especially considering realistic datasets contain millions of nodes and edges.

Several hardware accelerators were proposed to enhance the energy efficiency and performance of graph processing. Some of them have focused on optimizing memory access [148], others on improving the computational efficiency [149]. To further improve the system's efficiency beyond memory bandwidth limitation, near-memory DRAM-based computing was deployed in some previous works; e.g., TESSERACT [150] implements a vertex-centric programming model on top of Hybrid Memory Cube (HMC). To mitigate the communication overhead between different memory cubes, numerous solutions based on efficient graph partitioning [147], configurable interconnect [151], and batched-based communication [152] were provided.

Apart from DRAM-based accelerators, several CIM accelerators based on memristor devices were proposed. GraphR [91] is the first ReRAM-based graph processing accelerator. GraphR divides ReRAM into two parts; ReRAM memory, where the graph information is stored, and ReRAM engine, where computation is performed. Graphs are partitioned, and in each step, the ReRAM engines operate on several partitions in parallel. By exploiting the massive parallelism offered by ReRAM-based CIM as well as its unprecedented energy efficiency, they achieved up to 2 × and 9 × speedup and energy efficiency compared to GPU implementation. Similarly, GRAM [153] proposes a ReRAM-based accelerator for vertex-centric models. They also exploit the parallelism provided by CIM. Finally, unlike the works mentioned earlier, GraphSAR [154] takes into account graph sparsity and provides a CIM sparsity-aware design on top of memristor devices. In this approach, sub-matrices in which all elements are equal to zero are eliminated to improve efficiency in the presence of high data sparsity. Therefore, they incline to reduce the size of sub-matrices with the hope of eliminating more sub-matrices. This approach can partially eliminate the sparsity and has a high pre-processing overhead.

METAGENOMICS

In metagenomics, researchers study the behavior of many species altogether in a sample taken directly from an environment. The results of such a study help researchers to capture the complex relationship between different species without cultivating or isolating them individually in a very costly or yet impossible procedure for some species.

There are three key initial steps in a standard genome sequencing and analysis workflow [155]. The first step is the collection, preparation, and sequencing of a DNA sample in the laboratory. Modern sequencing machines are unable to read an organism's genome as a single complete sequence; instead, they generate shorter subsequences sampled randomly from the genome sequence [156, 157]. The second step is basecall-

ing, which converts the representation of the subsequences generated by the sequencing machine (e.g., images or electric current, depending on the sequencing technology [158]) into reads, which are sequences of nucleotides (i.e., A, C, G, and T in the DNA alphabet). In order to reproduce the complete genome sequence from the shorter read sequences, the third step, called read mapping, identifies potential matching locations of each read with respect to a known reference genome (e.g., a representative genome sequence for a particular species) [159, 160].

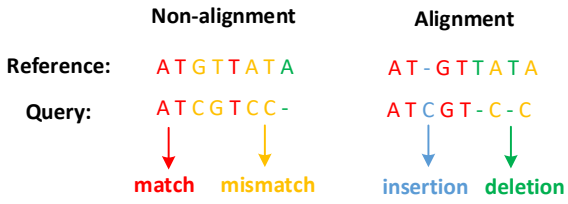


Figure 2.21: An example of a read mapping for non-alignment and alignment-based approach.

Read mapping can be classified into two approaches: Alignment-based and non-alignment-based (also known as heuristics). The alignment-based profilers are highly accurate (especially at the species rank). However, alignment-based profilers are very slow due to the high computational cost of their alignment. Rather, non-alignment-based profilers are less componential expensive at the expense of lower accuracy and providing less information. Figure 2.21 illustrates a read mapping step where differences between two DNA sequences are evaluated. In the case of alignment profilers, not only a similarity score is provided, but also it determines what kind of *edits* (substitutions, deletions, and insertions) and in which positions in a sequence might be happened. All these profilers induce large memory requirements for their data structures. For example, Kraken2 requires a minimum of 300 GB of memory for its reference data structure. Even for smaller and less complex reference databases, such as those in the food industry, more than 50 GB of memory is still required. The algorithms often used for read mapping are mainly simple bitwise logic and addition operations and can enjoy a high degree of parallelism provided by the underlying hardware. **Hence, exploiting CIM for this application can lead to extensive energy efficiency and speedup** due to reducing the data movement overhead and providing a high level of parallelism.

Accelerating Read Mapping:
There exists a large body of work on accelerating read mapping, which tends to follow two general directions: 1) non-filtering and 2) filtering approaches [155]. Neither of these two approaches is designed for processing in storage. Non-filtering approaches accelerate one or more non-filtering steps of read mapping (e.g., seeding and approximate string matching) using hardware accelerators. Examples of these accelerators include CIM architectures [161–163], ASICs [164], GPUs [165, 166], and FPGAs [167]. Filtering approaches accelerate the pre-alignment filtering step of read mapping. These works provide highly parallel read filtering heuristics that quickly eliminate dissimilar sequences before invoking computationally expensive alignment algorithms. Examples

of these accelerators include processing-near-memory architectures [168], FPGAs [169], GPUs [170], or traditional CPU-based acceleration [171]. GenASM [172] is a novel Near-memory DRAM-based approximate string-matching acceleration framework for genome sequence analysis. GenASM is a power- and area-efficient hardware implementation of our new Bitap-based algorithms. GenASM is a fast, efficient, and flexible framework for both short and long reads, which can be used to accelerate multiple steps of the genome sequence analysis pipeline. DARWIN [173] targets approximate string matching and sequence alignment. ApHMM [174] accelerate the Baum-Welch algorithms used in pHMM graphs. Helix [89] and KrakenOnMem [175] accelerate basecalling and metagenomics profiling on memristor-based systems. BLEND [176] proposes an efficient mechanism introduced to identify both exact-matching and highly similar seeds through a single lookup of their hash values, using a technique called SimHash and demonstrating its effectiveness in read overlapping and read mapping. Demeter [23] proposes a CIM-enabled architecture and a PCM-based accelerator to improve food profiling.

3

APPLICATION SPECIFIC DESIGN: NEURAL NETWORKS

This chapter focuses on our first application-specific design for Binary Neural Networks (BNNs) to realize the potential of CIM. Applications of BNNs are promising for embedded systems with hard constraints on energy and computing power. Unlike conventional neural networks using floating-point datatypes, BNNs use binarized weights and activations to reduce memory and computation requirements. Memristors, emerging non-volatile memory devices, show great potential as a target implementation platform for BNNs by integrating storage and compute units. However, the efficiency of this hardware highly depends on how the network is mapped and executed on these devices. In this chapter, we propose an efficient implementation of XNOR-based BNN to maximize parallelization. In this implementation, costly analog-to-digital converters are replaced with sense amplifiers with custom reference(s) to generate activation values. Besides, a novel mapping is introduced to minimize the overhead of data communication between convolution layers mapped to different memristor crossbars. This comes with extensive analytical and simulation-based analysis to evaluate the implication of different design choices considering the accuracy of the network. The results show that our approach achieves up to 5× *energy-saving and* 100× *improvement in latency compared to baselines.*

This chapter is based on [24].

3.1. INTRODUCTION

Neural Networks (NNs) are leveraged in a variety of applications [99–101]. With the growth of the network size for advanced applications, the implementation of NNs has become challenging, considering hardware limitations (e.g., BERT has around 110 million parameters [177]). Binary Neural networks (BNNs), where the weights and activation values are binarized (-1,+1), receive more attention from researchers due to their high model compression rate and simplified computations [136]. This is appealing for edge devices with hard constraints on memory capacity, computing resources, and energy budget. Although the computations are simplified, further improvement in the efficiency of BNN implementations relies on reducing the data transfer cost between memory and computing units (memory wall). Computation-in-Memory (CIM) and the unique characteristics of emerging non-volatile memories (memristors) [21, 178, 179] are promising candidates to deliver the next level of energy-efficiency implementation of BNNs. Hence, there is a need to design energy-efficient accelerators leveraging the notion of CIM to enable large-size networks for edge devices.

As discussed in Chapter 2, memristor-based CIM design not only reduces the overhead of data transfer but also can enhance the performance of VMM operation as a key kernel in BNNs. However, using memristors to operate on signed numbers (-1,+1) in BNN is challenging. From this perspective, existing works can be classified into hardware or algorithmic solutions. As a hardware solution, positive and negative values can be mapped to different memristors [137–139]. Other approaches consider one- [140] or two-column reference memristors [141] while converting the weights and activations to unsigned representation. In general, these approaches require more devices, increase design complexity, and reduce the energy/performance efficiency of the system. As an algorithmic solution, a signed VMM can be converted to XNOR operations [180] where the operands are unsigned (0,1). In this category, an accelerator is designed in which memristors are also used as an activation function [142]. This induces endurance, energy, and performance issues due to excessive memristor programming. To ensure the accuracy of XNOR operations against device variation, a new memristor crossbar structure based on differential sensing is proposed [12]. However, XNOR operations must be performed sequentially due to the sensing mechanism. All these overheads drive researchers to explore new mappings and implementations of BNNs to enhance their efficiency further for edge devices.

In this chapter, we present our methodology and design for efficient implementation of BNNs. The proposed mapping of operands for XNOR operations to the crossbar allows simultaneous crossbar row activation. This maximizes resource utilization on the crossbar and enhances performance. Moreover, we mimic the functionality of an Analog-to-Digital Converter (ADC) and the following digital processing, initially needed for this mapping, by only a Sense Amplifier (SA) with an adjusted reference. Furthermore, we minimize data communication between layers by proposing a novel mapping of the weights and activation values into the crossbar and its input buffer. We investigate the efficiency of our approach on different network structures in terms of accuracy, energy, and performance by developing our PyTorch-based

simulation platform [181]. The platform can mimic the behavior of the crossbar and allows for more characteristics and non-idealities to be integrated and explored for different networks. Our approach achieves close to 5× energy-saving and 100× improvement in latency compared to the state-of-the-art computation-in-memory designs at the cost of up to 4% accuracy loss. In this chapter, we present the following main contributions:

- An energy-efficient and highly parallel implementation of XNOR-based BNNs where the functionality of ADC and the required digital processing after that are modeled by a SA with an adjusted reference;
- An efficient mapping of the weights and activation values to improve data utilization and minimize the communication between network layers;
- An extensive analytical and simulation-based analysis where the proposed implementation behaves as an approximation to comprehend the implication of SA reference values on the accuracy of the design.

This chapter is organized as follows. Section 3.2 provides background binary neural networks. Existing accelerators for BNN are explained in Section 3.3. We discuss our proposal design in Section 3.4. In Section 3.5, we perform analytical analyses to elaborate more on the implications of the sensing scheme on accuracy. Finally, section 3.6 evaluates the design.

3.2. FUNDAMENTAL OF BINARY NEURAL NETWORKS

Designing larger networks and the ability to train them with advanced algorithms were the main drivers to enable neural networks for complex applications. However, implementing these networks in embedded platforms with limited storage and computation units is challenging, specifically in consideration of strict energy/performance constraints. Despite conventional neural networks with high precision datatypes, in BNNs, weights and activations are binarized to make the network extremely compact. Equation 3.1 shows a simple binarization rule that can be applied to both activations (input tensors) and weights where B_ω and B_I are the binarized weights and input tensors, respectively. The binarization not only saves on storage usage, but also reduces the expensive multiply-accumulate operations to simple additions.

$$B_\omega = \begin{cases} +1 & if\ \omega \geq 0 \\ -1 & if\ \omega < 0 \end{cases} \quad B_I = \begin{cases} +1 & if\ I \geq 0 \\ -1 & if\ I < 0 \end{cases} \tag{3.1}$$

Although binarization enhances the system's efficiency regarding memory usage, energy, and performance, it usually comes at the cost of accuracy loss compared to its high-precision counterpart. Therefore, using proper methods and algorithms to preserve the accuracy of the network as high as possible is essential. Each iteration of training a network can be divided into three steps: forward pass,

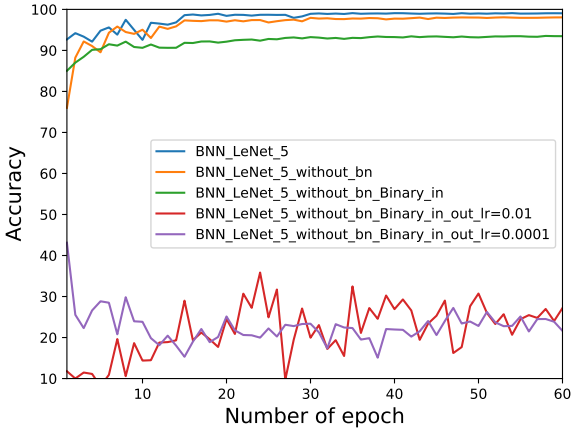


Figure 3.1: Accuracy of binarized LeNet5 network and the impact of input/output layer binarization as well batch normalization (bn) on accuracy loss using different learning rates (lr).

backward propagation, and parameter update. The weights during the forward pass and backward propagation are binarized. However, we need to use high-precision weights during parameter updates. Since parameter changes obtained by gradient descent are tiny, binarization ignores these changes, and the network cannot be trained [180, 182]. In addition, binarizing the input and output layers usually results in a huge accuracy loss. Figure 3.1 depicts the accuracy of the binarized LeNet5 network trained for the MNIST dataset. This clearly shows the impact of binarizing the input and output layers as well as batch normalization (bn) on the accuracy of the network. Batch normalization normalizes the contribution of layers' input. This helps to stabilize the learning processes during network training.

3.3. Overview of Existing BNN Implementations

To implement BNNs, besides using traditional systems (CPU, GPU, and FPGA) [183–185], computation-in-memory (CIM) accelerators based on emerging non-volatile memories (memristors) draw the attention of researchers. Memristor crossbar arrays are tailored to perform analog VMM with higher energy efficiency compared to their digital counterpart (CPU/GPU) [186]. Memristor devices usually can alternate between a few resistance levels (e.g., two levels), while more levels lead to reliability, stability, and accuracy degradation. Hence, BNN-based applications where the main kernel (VMM) is binarized are the promising targets to be implemented using memristor devices. A small-scale demonstration of a BNN on memristor devices is presented in [187], focusing mainly on device variation and its implication on accuracy. A new methodology is proposed in [188] to make the design more tolerant against device variation to be able to activate more word-lines and perform more computation at the same time. Based on Equation 3.1, BNNs require signed

representation, but negative numbers cannot be directly stored in memristors. Accordingly, existing BNN accelerators can fit into two categories based on how they address the problem.

- Hardware solutions: To deal with signed numbers, both weights and activation values can be represented as two vectors only holding absolute numbers; one holds positive and one holds negative values [137]. The two vectors created for both the weights and activation values are programmed to the corresponding memristors and sent to the input ports of a crossbar (select-line), respectively. Subsequently, the four possible partial results are computed and summed up in an analog manner. This requires a high number of memristor devices, which translates to low area and energy efficiency. In addition, since the proposed approach requires input current in both directions, the complexity of input drivers is increased. A similar approach is mapping positive and negative weights into different crossbars [138, 139]. In these works, ADC is exploited to compute the partial result when a BNN layer size is larger than the crossbar size. Then, the partial result from different crossbars is accumulated and given to an activation function. However, using ADCs imposes significant energy and area overhead on the system. Another solution is using one- [140] or two-column reference memristors [141] while the weights and activations are presented as {0,1}. In this design, the current flowing through the reference column(s) has to be mirrored equal to the number of columns in the crossbar. This increases the design complexity and energy consumption of the system. In addition, when a layer size cannot fit into a crossbar, it is critical to have a flexible referencing scheme to avoid accuracy loss. We discuss this more in Section 3.5.

- Algorithmic solutions: Binary multiply and accumulate operation can be replaced by the following sequence of operations: **XNOR, popcount**, and **post processing** [180]. As a result, the weights and activations for BNN can be presented as unsigned {0,1} values. This makes the implementation of BNNs on memristor crossbars simpler. Memristor-based content-addressable memory (CAM) structure can be used to implement binary XNOR operation and, in turn, BNNs [142]. In this design, the activation function is implemented by a memristor where its state determines the input value for the next layer. However, this suffers from an extremely high number of device programming, which causes challenges in terms of reliability, performance, and energy. An XNOR-based robust design to device imperfections is proposed using a differential sensing mechanism [12]. Due to the structure of the crossbar and the mapping of the weights, this design cannot exploit maximum parallelism in producing output values for each layer of BNN. This work is closest to our design and is considered a baseline. We elaborate more in the following section.

In conclusion, and considering the limitations and challenges of existing works, a highly parallel and energy-efficient BNN design is needed.

3.4. METHODOLOGY

In this section, we first explain the principles behind XNOR-based BNNs. This is used for both the implementation of BCIM and the baseline. Second, we discuss data mapping and execution of BNNs in the baseline [12]. Third, we present the new mapping and execution of BNNs in BCIM and compare it with the baseline. Fourth, we explain how the crossbar's input buffer, holding activation values, is managed to minimize data transfer between crossbars implementing different BNN layers.

3.4.1. MYLTIPLY-ACCUMULATE BASED ON XNOR OPERATION

The multiply-accumulate operation between two signed binarized vectors can be replaced by the sequence of 1) XNOR, 2) popcount, and 3) post-processing (Shift and Subtract) operations [180]. To achieve that, first, both vectors are converted from signed to unsigned, where '-1' is replaced by '0'. This is helpful considering memristor devices since it simplifies the mapping of weights to the crossbar without concern for negative values. Second, by applying Equation 3.2, the final value is obtained where *A*' is the unsigned representation of vector *A*. Popcount() returns the number of ones in a bitstream, and *'vector size'* is the length of the two vectors.

$$A * B = 2 * Popcount(A^{'} \odot B^{'}) - vector\ size \tag{3.2}$$

In the following, an example is provided to have better clarification. The result of the multiply-accumulate operation between vectors A and B from the traditional approach is:

$A = [1,-1,-1,1]\ B = [-1,1,1,1] \Rightarrow A * B = -2$

In the new approach, vectors $A^{'}$ and $B^{'}$ are created by converting A and B from signed to unsigned representation. First, we perform the XNOR operation between $A^{'}$ and $B^{'}$. Second, the result is given to the *Popcount* function. Third, the final processing, including Shift and Subtraction, is performed on the output of the Popcount function.

$A^{'} = [1,0,0,1]\ B^{'} = [0,1,1,1] \Rightarrow A^{'} \odot B^{'} = [0,0,0,1] \quad A * B = 2 * Popcount(A^{'} \odot B^{'}) - vector\ size = 2 * 1 - 4 = -2$

By applying the above method for BNNs, one vector can be considered as an activation vector ($A^{'}$) while another vector ($B^{'}$) holds the weights. The result is an activation value for the next layer. The process of generating the activation value for the next layer can be expressed as:

$$out_m = Sign(2 * \sum_{k=1}^{I} (in_k \odot \omega_{k,m}) - vector\ size) \tag{3.3}$$

where in_k represents the k^{th} activation value for the current layer; $\omega_{k,m}$ is the weight connecting the k^{th} activation value to the m^{th} output; and I is equal to the number of activation values of the current layer. The operator Σ performs as the Popcount function. Using this algorithmic solution to avoid representing negative data can

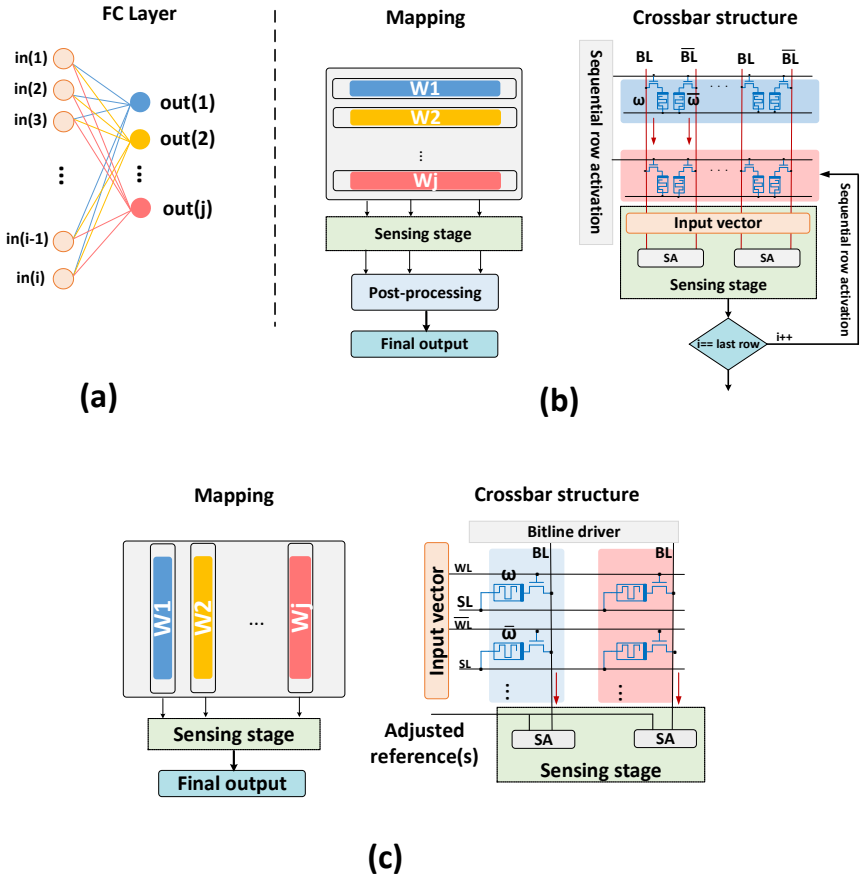


Figure 3.2: (a) An illustration of a fully connected layer (b) BNN implementation using differential sensing and sequential XNOR operation [12] (c) proposed design where massive XNOR operations are performed in parallel.

reduce the number of memristor devices 2× compared to a hardware solution [137]

In the following, we provide an example of a fully connected layer to explain how the baseline and BCIM map and execute BNNs based on the above methodology.

3.4.2. STATE-OF-THE-ART XNOR-BASED BNN

Figure 3.2(b) illustrates how a fully connected layer is mapped to a crossbar based on the approach proposed in [12]. The binary weights (ω) and their complements ($\overline{\omega}$), associated with each output channel (indicated by different colors), are programmed into one row of the crossbar. In order to compute the result for one output channel, first, its corresponding row is read. Second, the XNOR operation is performed with the (analog) value on the bit-lines of the crossbar and the input activation vector

(orange box in Figure 3.2(b)). This operation is done within the sensing stage by modifying the circuit of Sense Amplifiers (SAs). In this design, the complementary value for both weights and input vector is required to be able to perform the XNOR operation in the sensing stage. Finally, the output is given to the digital periphery to perform Popcount, Shift, and Subtract operations.

3.4.3. PROPOSED XNOR-BASED BNN IMPLEMENTATION

Figure 3.2(c) depicts the mapping of the weights and the crossbar structure in BCIM. All the weights (ω) and their complementary values ($\overline{\omega}$), corresponding to each output channel, are programmed in one column of the crossbar. We provide the input activation vector and its complement to the word-lines ($WL, \overline{WL}$) of the crossbar. Therefore, two memristors are allocated for each weight (ω and $\overline{\omega}$) and two world-lines for each activation value (WL and $\overline{WL}$). Hence, the XNOR operation between one element of the activation vector and the weight vector is computed as $in_k \odot \omega_{k,m} = in_k.\omega_{k,m} + \overline{in_k}.\overline{\omega_{k,m}}$ where in_k and $\overline{in_k}$ are the activation values provided to the WL and $\overline{WL}$, respectively. It should be noted that the summation between the two terms of the above formula is implemented with analog addition on the bit-line.

The vector resulting from XNOR operations on all the pair elements of activation and weights vectors should be passed through the Popcount function. This is indicated in Equation 3.3 by the Σ operator. Since each element contributes to the current flowing to the same bit-line, the analog sum of contributions represents the output of Popcount in the analog domain. In the naive approach, this analog value can be translated to the digital domain by using an Analog-to-Digital Converter (ADC). Then, we perform other operations (Shift and Subtraction) in the periphery of the crossbar. However, ADC is a power and area-hungry component [67]. Consequently, using this component not only reduces the energy efficiency of the design, but also has to be time multiplexed between several bit-lines, which in turn reduces the performance. However, according to Equation 3.3, the output of Popcount can be directly compared with a reference noted in Equation 3.4 to obtain the final output. Hence, the bit-line's analog output can be given to a SA with a customized reference to generate the output value. This also eliminates the remaining processing (Shift and Subtraction) in the digital periphery.

$$SA_{reference} = \frac{vector\ size}{2} \tag{3.4}$$

Based on this approach, we first maximized the number of parallel output activation values that can be computed for a BNN layer. All the bit-lines can be activated in parallel to compute the result for several output channels. Second, to avoid reducing the performance and energy efficiency by utilizing a high-resolution ADC, a simple analog SA with a customized referencing value is deployed. This not only performs the sign operation, but also omits extra digital processing in the periphery, thereby achieving considerable energy and performance improvement. Third, in this method, when vectors cannot fit into one column of the crossbar, they

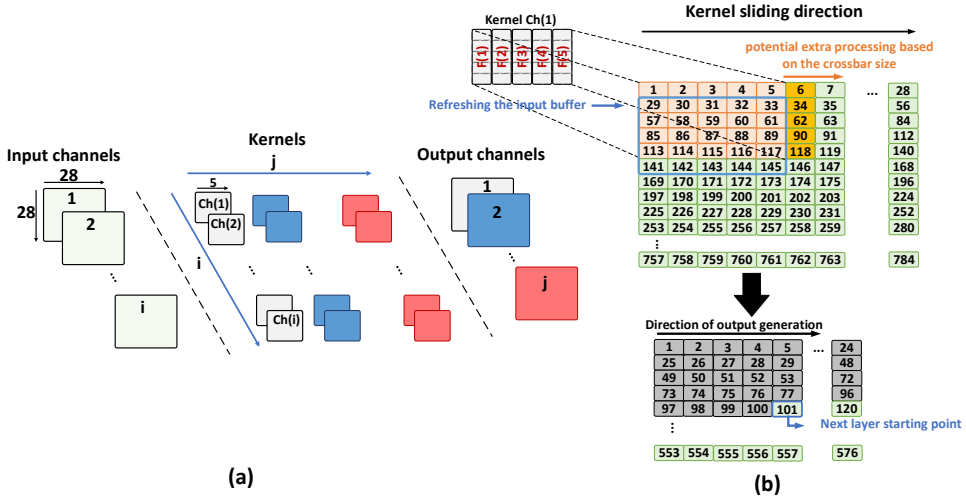


Figure 3.3: (a) Example of a CNN layer and (b) details of a convolution operation with 5×5 and 28×28 kernel and input size.

have to be broken and mapped to several columns. This may lead to approximate computing. In Section 3.5, we scrutinize this interesting scenario analytically.

3.4.4. EFFICIENT DATA MOVEMENT

Data movement between the BNN layers may influence the performance and energy of the system [67], but is often overlooked by the existing works. In this subsection, we focus on how the data should be transferred from one convolutional layer to the next one to minimize the number of transactions and the size of a buffer placed between layers. This approach can be utilized for both binary and non-binary datatypes.

Figure 3.3(a) depicts an example of a convolution layer where the kernel matrix is convolved into the *"i"* input channels to generate data for the *"j"* output channels. In this example, the input size for each channel and the kernel size are 28×28 and 5×5, respectively. Figure 3.3(b) illustrates the details of the convolution operation where each kernel slides on a corresponding input channel to produce the partial result. The kernels are programmed to the crossbar while the data of input channels corresponding to the current operating window (highlighted by light orange) are buffered and sent to the word-lines of the crossbar. When the operating window slides, the data has to be sent and reorganized in the buffer to be matched to the weights of the kernel programmed into the crossbar. However, bringing the whole data again for the following operating window is not an efficient way since most of it already exists in the input buffer of the crossbar from the previous operating window.

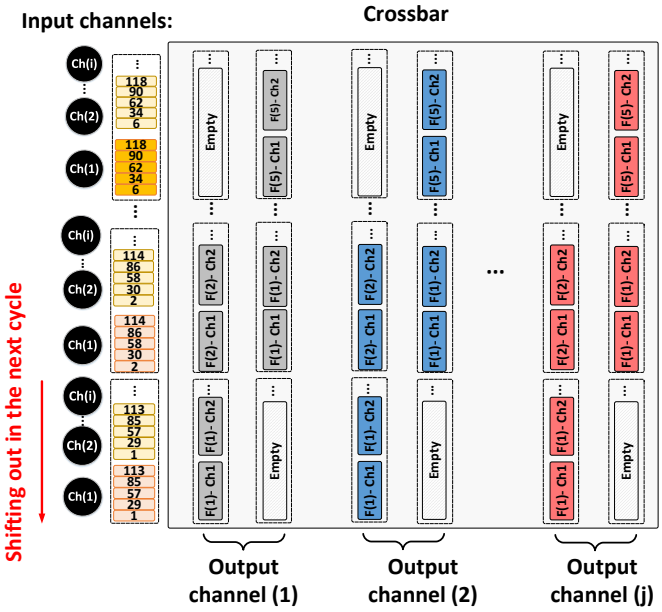


Figure 3.4: Mapping of the activation values to the input buffer and kernels to the crossbar based on the proposed approach to minimize data transmission between layers by only streaming the newly computed activation values into the input buffer.

To provide better data utilization and reduce the number of transactions, Figure 3.4 demonstrates an efficient mapping of kernels in the crossbar as well as activation value in the input buffer. In this approach, the kernels and the input data within the operating window are sliced into columns. The same columns for different input channels are packed together and placed in the input buffer. The next columns are stacked on top of each other as highlighted by the light orange color in Figure 3.4. The kernels are also treated the same way. By doing that, when the operating window slides to the right (assuming stride is one), the left-most columns for all the input channels are shifted out and new data corresponding to the right-most columns are streamed into the buffer. There is no need to change the mapping of the kernels in the crossbar and they always reside in front of the right inputs. When the operating window reaches the last columns, it has to be shifted down and start from the most left column again. Therefore, the input buffer is refreshed and filled with data highlighted by the blue window in 3.3(b). As a result, maximum data is utilized when the operating window slides while the input buffer can be implemented as simply as possible.

In order to maximize the performance, we can exploit parallelization and pipelining. In case the crossbar dimension is large enough, the computation for the current and next operating windows can be performed in parallel. As illustrated in

Figure 3.3(b) and 3.4, an extra column (highlighted by bright orange) required for the next operating window is placed into the input buffer of the crossbar. Besides, we have to consider another column in the crossbar to be able to generate the value for both operating windows simultaneously. It has to be taken into account that this extra input set should not contribute to the computation of the current window. Therefore, the memristors located in the first column and in front of this extra input set should be programmed to logic value '0'. It is worth mentioning that the kernels for other output channels are programmed to different columns of the crossbar to maximize parallelization. However, in case the crossbar has a lower number of columns, we need to deploy more crossbars to avoid an excessive number of reprogrammings. Besides parallelization, the same pipelining approach presented in [67] can be applied in this work. Depending on the kernel size of the next layer in the network, when enough elements are produced for the output channels of the current layer, the operation can be started for the next layer.

3.5. INTRA-LAYER ACCURACY ANALYSIS

In Section 3.4, the proposed implementation was presented where a single SA can generate the activation value for the next BNN layer (see Figure 3.2). However, if the weights that are supposed to be in a single column of a crossbar cannot fit into it, they have to be split and mapped to more columns. In other words, if there are not enough memristors in a column of a crossbar to store a kernel (e.g., the blue kernel in Figure 3.2(c)), this kernel has to be broken into several parts each mapped to different columns. Therefore, the final activation value has to be calculated from the intermediate activation values obtained from different sets of columns. This is where inaccuracy is injected into the network with a particular probability distribution.
In the following, the ideal situation is formulated where the crossbar size is equal to or greater than the vector size. $\overrightarrow{A}$ and $\overrightarrow{B}$ are the two input binary vectors, $\overrightarrow{R}$ is the result of XNOR operation between the two input vectors, and $\Sigma(\overrightarrow{R})$ produces the output of Popcount function on the binary vector $\overrightarrow{R}$.

Vector size $= v$, *Crossbar size* $= C$, and $C \geq v$

input 1: $\overrightarrow{A}$, input 2: $\overrightarrow{B}$

$\overrightarrow{R} = \overrightarrow{A} \odot \overrightarrow{B}$

$$out_{golden}(\overrightarrow{R}) = \begin{cases} 1 & \text{if} \quad \Sigma(\overrightarrow{R}) > v/2 \\ 0 & \text{otherwise} \end{cases}$$

In case the crossbar size is not big enough, the formulation is changed as presented below. As an example, we assume the crossbar size is half of the vector size. Therefore, each vector has to be split into two parts and mapped to two columns of the crossbar.

Vector size $= v$, *Crossbar size*: $C = v/2$

input 1: $\overrightarrow{A}|_0^{v/2}, \overrightarrow{A}|_{v/2}^{v}$ where $\overrightarrow{A} = [\overrightarrow{A}|_0^{v/2}, \overrightarrow{A}|_{v/2}^{v}]$

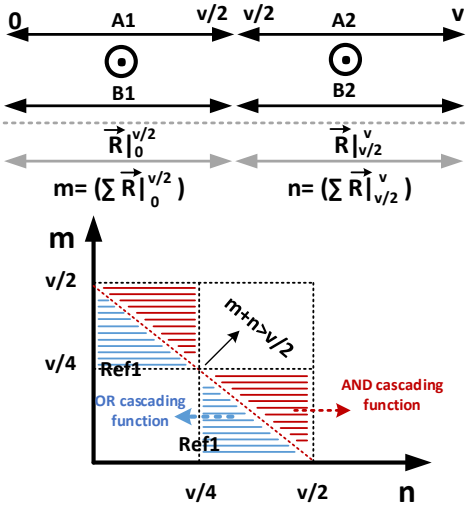


Figure 3.5: Illustration of the regions where logical *AND* and *OR* cascading functions inject inaccuracy into the network.

input 2: $\vec{B}|_0^{v/2}, \vec{B}|_{v/2}^{v}$ where $\vec{B} = [\vec{B}|_0^{v/2}, \vec{B}|_{v/2}^{v}]$

$$\vec{R}|_0^{v/2} = \vec{A}|_0^{v/2} \odot \vec{B}|_0^{v/2} \quad \vec{R}|_{v/2}^{v} = \vec{A}|_{v/2}^{v} \odot \vec{B}|_{v/2}^{v}$$

$$out_{p1}(\vec{R}|_0^{v/2}) = \begin{cases} 1 & \text{if} \quad \Sigma(\vec{R}|_0^{v/2}) > (v/2)/2 \\ 0 & \text{otherwise} \end{cases}$$

$$out_{p2}(\vec{R}|_{v/2}^{v}) = \begin{cases} 1 & \text{if} \quad \Sigma(\vec{R}|_{v/2}^{v}) > (v/2)/2 \\ 0 & \text{otherwise} \end{cases}$$

Since we mapped the vector into two columns, two intermediate activation values (out_{p1}, out_{p2}) are obtained. The final value depends on the *cascading function*, which receives intermediate activation values (out_{p1}, out_{p2}) as input and produces the final activation value. This function can be a simple logical *AND* or *OR* function. The following is an example of AND $(\wedge)$ cascading function.

$$out(\vec{R}|_{v/2}^{v}, \vec{R}|_0^{v/2}) = out_{p2}(\vec{R}|_{v/2}^{v}) \wedge out_{p1}(\vec{R}|_0^{v/2})$$

In the case of logical *AND* as an example, the following conditions show the scenarios where the output of the cascading function differs from the golden output. This is also illustrated in Figure 3.5. The y and x axes are the output of Popcount (Σ) obtained from the result of the first $(\vec{R}|_{v/2}^{v})$ and second parts $(\vec{R}|_0^{v/2})$ of the output vector. The red and blue regions indicate inaccurate results by the *AND* and *OR* functions.

$out(\overrightarrow{R}|_0^{v/2}, \overrightarrow{R}|_{v/2}^{v}) \neq out_{golden}(\overrightarrow{R})$ if:

$$\begin{cases} \Sigma(\overrightarrow{R}|_0^{v/2}) + \Sigma(\overrightarrow{R}|_{v/2}^{v}) = \Sigma(\overrightarrow{R}|_0^{v}) > v/2 \\ \Sigma(\overrightarrow{R}|_0^{v/2}) < v/4 \quad \vee \quad \Sigma(\overrightarrow{R}|_{v/2}^{v}) < v/4 \end{cases}$$

According to the aforementioned conditions, the output of AND cascading function does not generate the expected results only if 1) the final activation value is expected to be '1' ($\Sigma(\overrightarrow{R}|_0^{v}) > v/2$) and 2) one of the intermediate activation values regarding the partial output vectors ($\overrightarrow{R}|_0^{v/2}$ or $\overrightarrow{R}|_{v/2}^{v}$) is '0'. Hence, the final result, which is generated using logical AND between the output of the two intermediate activation values from $\overrightarrow{R}|_0^{v/2}$ and $\overrightarrow{R}|_{v/2}^{v}$ vectors would be '0'. We illustrate these two conditions in Figure 3.5. The region above the $m+n=v/2$ line satisfies the first condition. In this region, the data points that fall into the two triangles, highlighted in red, meet the second condition. It should be noted that the condition where the expected final activation value is '0', but both partial activation values would be '1' never happens.

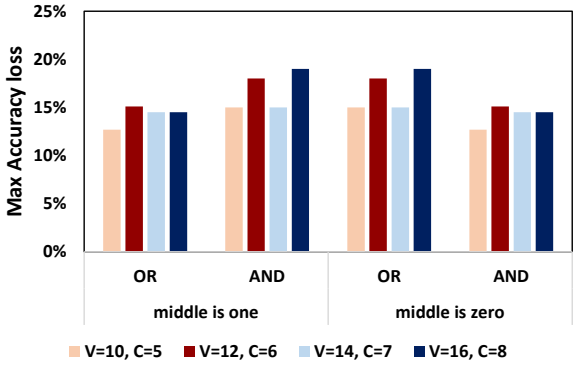


Figure 3.6: Maximum accuracy loss simulated for all possible input vectors for different vector sizes (V), crossbar size (C), and cascading functions.

The combinations of input vectors for a given (m,n) in these regions (blue for OR and red for AND cascading function in Figure 3.5) is calculated based on Equation 3.5. m and n are the outputs of Popocount function for the two partial output vectors resulting from XNOR operations. Accordingly, Equation 3.6 calculates all the possible combinations of input vectors that fall into the 'Solution Set'. The solution set for AND cascading function is highlighted in red in Figure 3.5. According to these two equations, Figure 3.6 depicts the maximum accuracy loss for two cascading functions considering two boundary conditions. The boundary condition determines the output of SA in case the data is the same as the reference. This is done by generating all the combinations of input sets to verify the Equation 3.5. We observe that the accuracy loss does not have considerable changes over vector sizes as the relative area associated with inaccurate region remains the same (Figure 3.5). It should be noted that this accuracy loss in Figure 3.6 should not be confused with the accuracy of an entire BNN.

$$N(m,n) = ({}_mC_{v/2} * 2^m * 2^{v/2-m}) * \\ ({}_nC_{v/2} * 2^n * 2^{v/2-n}) = 2^v * ({}_mC_{v/2} * {}_nC_{v/2}) \tag{3.5}$$

$$TN_{(AND)} = \sum_{(m,n) \in Solution\ Set} N(m,n) \tag{3.6}$$

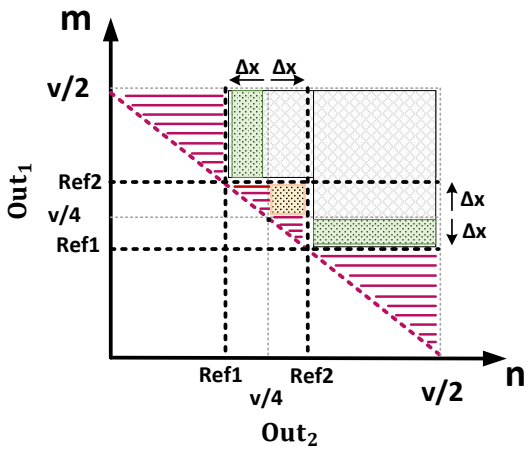


Figure 3.7: Illustration of the scenario where there are two references for each partial output vector. The value of references (or Δx) should be defined in a way that the cascading function covers more area above $m+n=v/2$ line.

In the aforementioned example (see Figure 3.6), the activation value for each partial output vector ($\overrightarrow{R}|_0^{v/2}$ and $\overrightarrow{R}|_{v/2}^{v}$) are obtained by only employing one reference (i.e., $v/4$). In order to reduce the accuracy loss, more references can be considered. This leads to more intermediate results, which provide us with more information as well as the flexibility to have advanced cascading functions. However, we should take into account that adding references increases the hardware complexity of SA. Next, we investigate a scenario where SAs have two references.

Two references: Figure 3.7 illustrate the scenario where SAs have two references. This figure provides insight into the impact of references and their value on accuracy. Similar to Figure 3.5, m and n are the outputs of the Popocount function for the two partial output vectors resulting from XNOR operations. The goal is to determine where to put the references (or determine Δx) to minimize the accuracy loss (or minimize highlighted red region in Figure 3.7). We assume the references are placed symmetrically around the center point to simplify the analysis. As mentioned before, we need a cascading function to receive intermediate activation values (SA_{out1}, SA_{out2}), generated by the SAs, and produce the final activation value. In the following, we describe one example of a cascading function.

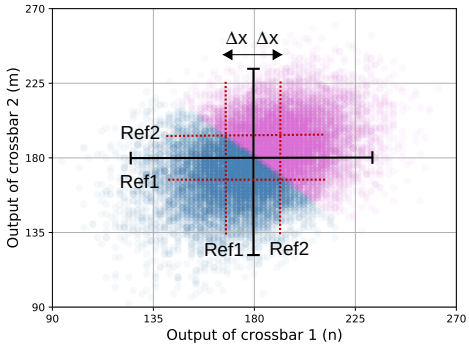


Figure 3.8: Data points for two partial output vectors obtained from a real network. The size of each vector ($v/2$) is 360. The figure clearly shows a non-uniform distribution of data points.

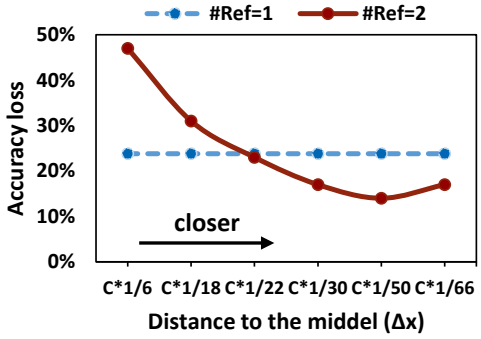


Figure 3.9: Accuracy loss based on the value of Δx presented relative to the crossbar size (C).

$$F= [(SA_{out2} > Ref2) \wedge (SA_{out1} > Ref1)] \vee [(SA_{out2} > Ref1) \wedge (SA_{out1} > Ref2)]$$

According to the above cascading function, the final activation value would be 1 only if one of the outputs is higher than Ref_2 while the other one is higher than Ref_1. In Figure 3.7, the area covered by this function is highlighted in gray and green. As we can see, this function cannot cover the entire area above the $m+n=v/2$ line. The uncovered part is highlighted in red. Compared to the scenario where we have only one reference (see Figure 3.5), some new regions are covered (highlighted in green) while one region is left out (highlighted in yellow). Depending on the value of Δx, the size of these regions changes. If the data points were distributed uniformly, we could easily obtain the optimum Δx where the area covered by this cascading function is maximum. However, according to Equation 3.5 (or Figure 3.8), the data is not uniformly distributed. Hence, simulation can help to find the optimum Δx. Figure 3.9 shows the accuracy of this cascading function over different Δx. In this

figure, the value of Δx is presented relative to the crossbar size ('C'), which is equal to $v/2$. According to this figure, using two references can lead to better accuracy than one reference (dashed blue line) if we can find the optimum values for the references. In addition, it is worth clarifying that accuracy results in Figure 3.9 are not the final BNN accuracy.

It should be noted that when we have two references, putting one reference in the center does not make sense since the area covered by the second reference would be either a subset or superset of the first reference. However, this is not the case when we have three references. In the following, we elaborate on the scenario where we have three references.

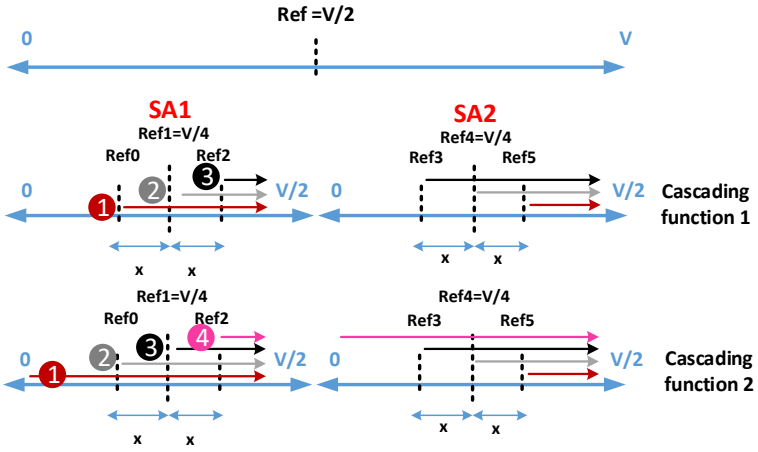


Figure 3.10: Illustration of two cascading functions where two auxiliary references are added to the main reference.

Three references: Figure 3.10 presents an example where three references (Ref0, Ref1, Ref2) are considered to generate an activation value. In the ideal scenario where there is no need to split the vectors, the reference, obtained from Equation 3.4, is equal to $v/2$. However, for the scenario where we have to split the vector into two parts, the reference for the two partial output vectors, based on the same equation, should be $v/4$. This is called primary reference and indicated by "Ref1" in Figure 3.10. In this example, next to the primary reference, we utilize two more -auxiliary references- to improve accuracy. Next, we investigate the implication of the number of auxiliary references as well as their actual values on accuracy loss.

Considering the aforementioned scenario where three references are employed, we can produce three intermediate values for each of the partial output vectors. Hence, the final activation value should be decided based on these six binary values. In the following, we describe two possible cascading functions to produce the final activation value. They are also illustrated in Figure 3.10.

Prime implicant of cascading function 1:

F1= $[(SA_{out2} > Ref5) \wedge (SA_{out1} > Ref0)] \vee [(SA_{out2} > Ref4) \wedge (SA_{out1} > Ref1)] \vee [(SA_{out2} > Ref3) \wedge (SA_{out1} > Ref2)]$

Prime implicant of cascading function 2:
F2= $[(SA_{out2} > Ref5)] \vee [(SA_{out1} > Ref2)] \vee [(SA_{out2} > Ref3) \wedge (SA_{out1} > Ref1)] \vee [(SA_{out2} > Ref4) \wedge (SA_{out1} > Ref0)]$

The numbers in Figure 3.10 indicate the different conditions where the cascading function produces logic 1 as the final activation value. As an example, the first cascading function comprises three conditions, where meeting each can set the final activation value to 1. Each number in Figure 3.10 illustrates one term in the prime implicant of cascading function 1. These are based on the fact that the summation of two Popcount functions (Σ) obtained from two output vectors should be greater than half of the original vector size (Equation 3.4). This function always sets the activation value to one accurately (true positive), but it misses to set it to one in some cases (false negative). Considering that, the second cascading function makes the conditions more relaxed. The probability of accuracy loss for these two functions is computed in the following.

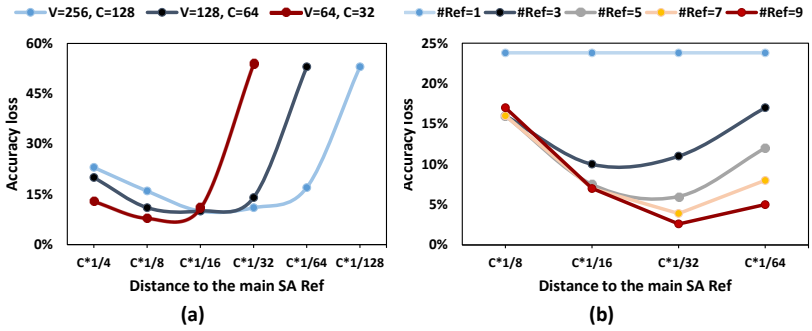


Figure 3.11: (a) Accuracy loss based on the distance of two auxiliary references to the main reference (b) effect of the number of auxiliary references on accuracy.

$P_{Loss}(F1) =$
$\mathbf{P}([SA_{out2} > Ref5 \wedge SA_{out1} < Ref0] \wedge [SA_{out2} + SA_{out1} > v/2]) + \mathbf{P}([SA_{out2} < Ref3 \wedge SA_{out1} > Ref2] \wedge [SA_{out2} + SA_{out1} > v/2]) + \mathbf{P}([Ref4 < SA_{out2} < Ref5] \wedge [Ref0 < SA_{out1} < Ref1] \wedge [SA_{out1} + SA_{out2} > v/2]) + \mathbf{P}([Ref3 < SA_{out2} < Ref4] \wedge [Ref1 < SA_{out1} < Ref2] \wedge [SA_{out1} + SA_{out2} > v/2])$

$P_{Loss}(F2) =$
$\mathbf{P}([SA_{out2} > Ref5] \wedge [SA_{out2} + SA_{out1} < v/2]) + \mathbf{P}([SA_{out2} > Ref5] \wedge [SA_{out2} + SA_{out1} < v/2]) + \mathbf{P}([Ref4 > SA_{out2} > Ref3] \wedge [Ref2 > SA_{out1} > Ref1] \wedge [SA_{out2} + SA_{out1} < v/2]) + \mathbf{P}([Ref5 > SA_{out2} > Ref4] \wedge [Ref1 > SA_{out1} > Ref0] \wedge [SA_{out2} + SA_{out1} < v/2])$

An important parameter that has a remarkable impact on the accuracy loss is the

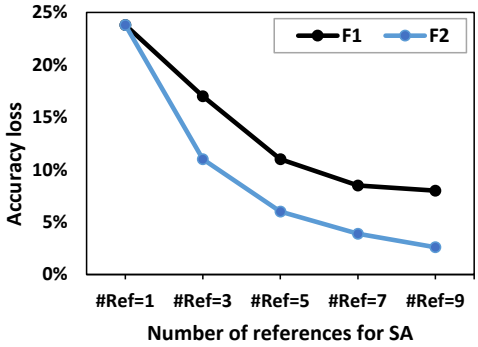


Figure 3.12: Impact of the two cascading functions illustrated in Figure 3.10 on accuracy loss.

distance of auxiliary references to the main reference (*"x"* in Figure 3.10). This is quite dependent on the distribution of data. Hence, the designer can analyze the network and, based on that, find the proper value for the references where the accuracy loss is minimized. Figure 3.11(a) demonstrates the impact of this parameter for cascading function 2 assuming a normal distribution. This is presented for different crossbar sizes ("*c*"). The distance to the main reference is shown relative to the crossbar size. The figure indicates the importance of the values for the references and how considerably they can change the accuracy loss. Another important parameter is the number of references. The implication on accuracy can be comprehended from Figure 3.11(b). It is observed that by keep adding more references, an improvement in accuracy is reduced while more complexity is added to the hardware. Finally, the impact of the cascading functions on accuracy is evaluated in Figure 3.12 over a different number of references. The same two methods presented in Figure 3.10 are also used for the situation where we have more than three references. The figure indicates that choosing a proper function can help the accuracy of the system remarkably.

3.6. EVALUATION

3.6.1. SIMULATION SETUP

Our simulation results are obtained by creating our PyTorch-based platform [181]. This platform is able to evaluate the accuracy, energy, and latency of different networks containing binarized and non-binarized layers. The software is written in a modular way to flexibly change network structure as well as different circuit-level parameters. The system runs at a clock frequency of 1GHz. The width of the databus transferring data between the crossbars is 32 bits. This is required for communication between layers. Based on the 32nm technology node, transferring data to store it in an input buffer consumes 5mW [14, 16]. The energy and latency number of the "Shift and Add" unit required for non-binarized layers taken from

Table 3.1: Typologies of the BNNs and their software accuracy

Name	Topology	Dataset	Accuracy
LeNet-5	5x5,6 - 2x2 Pool - 5x5,16 - 2x2 Pool - FC(120) - FC(84) - FC(10)	MNIST	98%
CNN-1	5x5,5 - 2x2 Pool - FC(720) - FC(70) - FC(10)	MNIST	97%
CNN-2	7x7,10 - 2x2 Pool - FC(1210) - FC(1210) - FC(10)	MNIST	98%
MLP-S	FC(784) - FC(500) - FC(250) - FC(10)	MNIST	97%
MLP-M	FC(784) - FC(1000) - FC(500) - FC(250) - FC(10)	MNIST	98.2%
MLP-L	FC(784) - FC(1500) - FC(1000) - FC(500) - FC(10)	MNIST	98.4%
AlexNet	11x11,96 - 3x3 Pool/2 - 5x5,256 - 3x3 Pool/2 - 3x3, 384 - 3x3,384 - 3x3,256 - 3x3 Pool/2 - 3x3 AvgPool/2 - FC(4096) - FC(4096) - FC(10)	(enlarged) CIFAR-10	80%

[14]. In all the simulations, the crossbar size is 512×512 [189]. We use an analytical model based on a small ReRAM memristor prototype and extend the memory to the required size. The LRS and HRS for the memristors are 5k and 1G, respectively. The read voltage is 0.2V and the latency of the crossbar to charge the bit-lines is considered to be 10 ns. The model is acquired from the results of the EU project MNEMOSENE [19].

The specification of the sensing mechanism is taken from [190]. The energy per ADC read is 12 pJ, and its latency is 3 ns. Besides, the energy of SA is assumed to be 10 fJ with 1 ns latency. In case our SA needs more references (e.g., 3 references), its energy and latency get increased linearly by the number of references [66]. A maximum of 3 references for a SA are considered in the simulations. The energy and latency numbers are parameterized in the simulation platform and can be changed based on different circuit designs.

Our benchmark (MlBench) comprises 7 BNNs for machine learning applications. The structure of each network is listed in Table 3.1. LeNet-5, CNN-1, CNN-2, and AlexNet are convolutional networks, and MLP-S/M/L are multilayer perceptrons (MLPs) with different network scales [110]. We use MNIST and CIFAR-10 datasets to evaluate our networks. The input images for CIFAR-10 are enlarged to 256×256 required for AlexNet. We compare our design with a recent work published in one of the leading journals in this field [12]. For this work, we instantiate the digital post-processing units (popcount) for every 16 columns of the crossbar instead of sequentially operating over all the columns (see Figure 3.2(a)). This diminishes the latency overhead of digital processing for the baseline.

3.6.2. RESULT AND DISCUSSION

In the following, first, we present the total accuracy loss for different networks. Second, we evaluate the design in terms of energy and performance and elaborate more on our observations.

Accuracy analysis

Figure 3.13 depicts the accuracy loss using our proposed approach compared to the software implementation. The figure presents the results for the benchmarks considering different cascading functions (see Figure 3.10). Depending on the size of

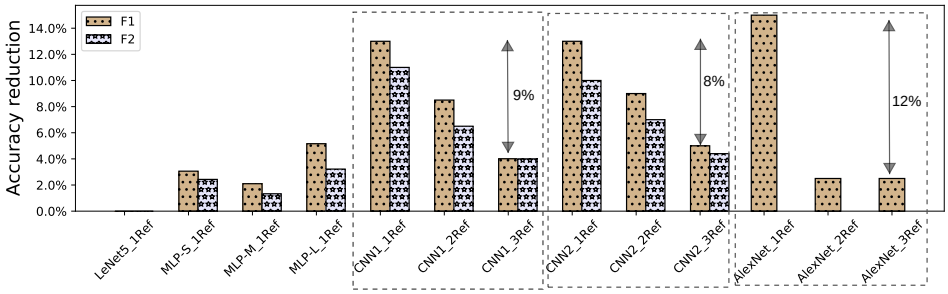


Figure 3.13: Accuracy reduction for different network structures due to the crossbar size limitation and breaking the vectors over more crossbars. F1 and F2 are two cascading functions. For SAs with one or two references cascading functions are 'AND' and 'OR'. For SAs with three references cascading functions are described in Figure 3.10.

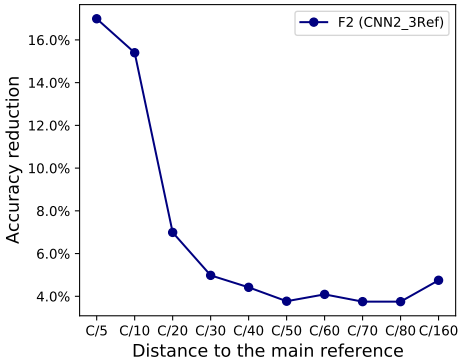


Figure 3.14: Impact of auxiliary references and their distance from the main reference on accuracy loss. The simulation is performed for the CNN2 network employing cascading function F2 which is described in Figure 3.10.

the layers in a network, we can see whether we have an accuracy loss or not.
1) Small-size network layers (LeNet5): Since the size of layers in the LeNet-5 network is within the range of crossbar size, no accuracy loss is observed. To clarify more, we can consider one of the fully connected layers -FC(84)- where there are 84 output activations. This layer receives 120 inputs. Hence, the number of memristor devices we need for a single column in order to accommodate this vector is 120×2 (see Figure 3.2(c)). Therefore, considering the crossbar size, there is no need to break the vector and reduce the accuracy.
2) Medium- and large-size network layers: The rest of the networks used in our simulations have larger layers to fit in one crossbar. Therefore, each layer has to

be broken and mapped into several crossbars. This is where inaccuracy is injected into the network. Figure 3.13 shows that accuracy reduction using SA with only 1 reference is up to 14%. This accuracy loss is much less for MLP networks. Considering CNNs (CNN1, CNN2, and AlexNet), adding two more references to the SA can improve the accuracy by up to 12%. This means that only using three references in the SA can provide a decent accuracy loss of around 2%. It is worth mentioning that in case we want to perform computation "precisely", an ADC should be used in the design. One can interpret an 8-bit ADC as a SA with 256 reference levels. Therefore, Figure 3.13 shows that with a smart selection of three references out of those 256 references, only 2% accuracy loss can be observed. In the case of AlexNet, since the size of layers is extremely large, layers are broken and mapped into more crossbars. Therefore, more options are available on how to perform the cascading function. The detail of the cascading function used for AlexNet can be found in [181].

Besides the number of references, another important parameter that can have a remarkable impact on accuracy is the actual value of references. Figure 3.14 depicts the implication of positioning the references on the accuracy loss. The simulation is performed for the CNN2 network with three references by changing the distance of auxiliary references to the main reference ("x" in Figure 3.10). The distance is relative to the crossbar size ("C"). Placing the references far from or too close to each other reduces their efficiency in eliminating the cases where inaccurate activation values are generated. Therefore, the designer should find the optimal value for the references by profiling the network.

Energy analysis

Figure 3.15 presents the energy numbers of different networks for the classification of one input image. The figure shows the energy number for 1) the baseline, 2) BCIM, and 3) the design where we want to do exact computing using ADC. The result indicates BCIM can achieve around 40% energy improvement compared to the baseline. Besides, BCIM can reduce the energy 5× compared to the design where exact computing is performed. In addition, we show the energy breakdown of AlexNet as well as its total energy in Figure 3.16. It is clear that energy consumption has a strong correlation with layer size and the amount of computation that has to be done in a layer. In addition, although the last layer is not binarized, its contribution to the total energy is very limited. This is due to the size of this layer compared to other layers (impact of output binarization on accuracy shown in Figure 3.1).

Comparing BCIM with the baseline in terms of energy, there are two factors that contribute to this energy improvement. First, the number of transactions required between the layers (or crossbars) is 3 times less in BCIM compared to the baseline. We present this relative comparison in Figure 3.17(a). This improvement is due to the mapping of activations and weights into the input buffer and the crossbar, respectively (see Figure 3.3). Second, the number of times SA is activated in BCIM is less than the baseline. We show this in Figure 3.17(b). It is worth mentioning that in both designs, the major contributor to the energy is the crossbar rather than the periphery or the data transfer between the crossbar. Hence, this is the reason BCIM

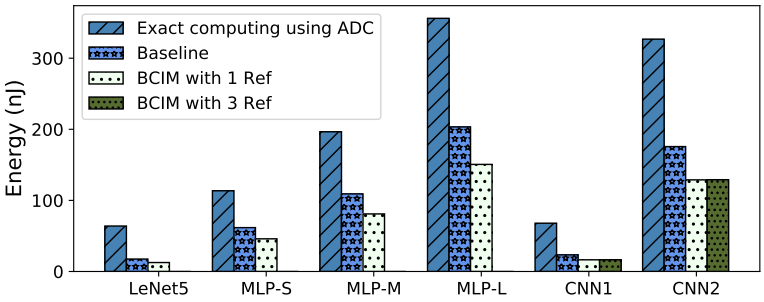


Figure 3.15: Energy consumption of BCIM compared to the baseline as well the design performing precise computing using ADC.

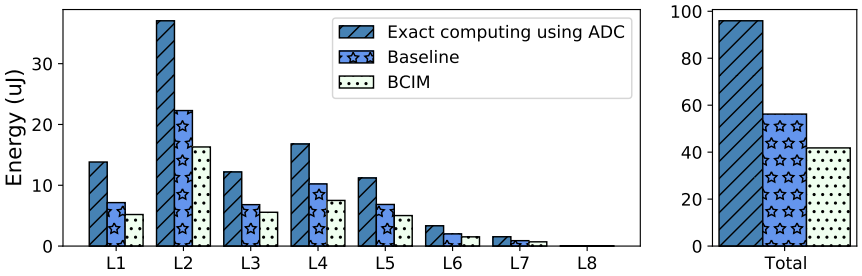


Figure 3.16: Energy breakdown as well as total energy consumption of AlexNet for BCIM, baseline, and exact computing designs.

archives marginal improvement in terms of energy compared to the baseline. In the future, if the energy consumption of memristor devices improves, then these two factors will play a major role in the energy.

In case we aim for exact computing using ADC, the total energy consumption significantly increases. The Major contributor to this energy rise is the costly ADC component. As mentioned in the Simulation setup section, ADC consumes 12 pJ per conversion. Considering the number of conversions required per crossbar activation, this imposes significantly more energy consumption on the system than a SA with around 10 fJ energy per sensing.

Latency analysis

Figure 3.18 shows the relative latency improvement for our different networks normalized to the baseline. BCIM achieves up to 100× improvement compared to the baseline. Besides, compared to the design where we perform exact computing, BCIM improves the latency by more than 3×.

There are two major factors involved in BCIM latency improvement compared to the baseline. We explain them in the following.

1) Mapping of weights into the crossbar: As illustrated in Figure 3.2(b), due to the way weights are mapped to the crossbar as well as the way computation is performed

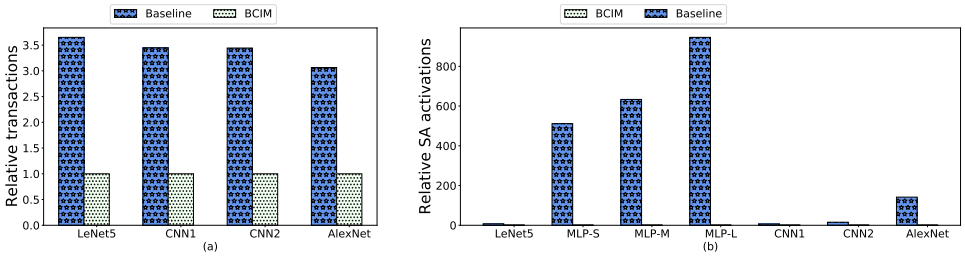


Figure 3.17: (a) Number of transactions between crossbars and (b) number of SA activation for entire networks (normalized to BCIM).

in the baseline, the output activation values are produced sequentially. This means at each time step, one row of the crossbar associated with one output value is activated. However, because of the weight mapping and the way we performed XNOR operations, the output activation values are produced in parallel (see Figure 3.2(c)). This improves the total latency of the network considerably. Figure 3.19(a) presents the contribution of different parts of the design for the baseline to the total latency. As we can see, the crossbar has a major contribution. However, in BCIM, by changing the mapping of the weights, the contribution of the crossbar was reduced significantly. We can observe this in Figure 3.19(b). In addition to the crossbar, the total latency of the periphery is high in the baseline, as we can see in Figure 3.19(b). This is because every time a crossbar row is activated, the sensing and all the digital peripheries should be conducted.

2) Mapping of activation value into the input buffer: Figure 3.19(b) shows that by changing the weight mapping and reducing the contribution of crossbar and periphery in the total latency, data communication becomes dominant in most cases. Hence, this is critical to have an optimized mapping of activation value into the input buffer in order to minimize the data communication overhead. As we can see in Figure 3.19(b), even after the optimization of the input buffer, the communication overhead is still dominant.

BCIM improves the latency by more than 3× compared to the exact computing design (see Figure 3.18). First, ADC imposes more latency in the periphery compared to SA in order to perform a conversion. Second, due to the high area consumption of ADC compared to a SA, more bit-lines in the crossbar should be shared by an ADC. Hence, this also increases the latency of this design.

Figure 3.19 presents the contribution of three major latency consumers (transaction, crossbar, and periphery) for (a) the baselines, (b) and BCIM. Considering the baseline, crossbars contribute more to the latency than other contributors. This is due to the large amount of sequential computation that has to be performed in the crossbar. However, in the case of BCIM, we can observe that the crossbar has the highest latency only for CNNs. In a convolution layer, a kernel (mapped to a crossbar) has to slide over the entire input data in order to produce output activation

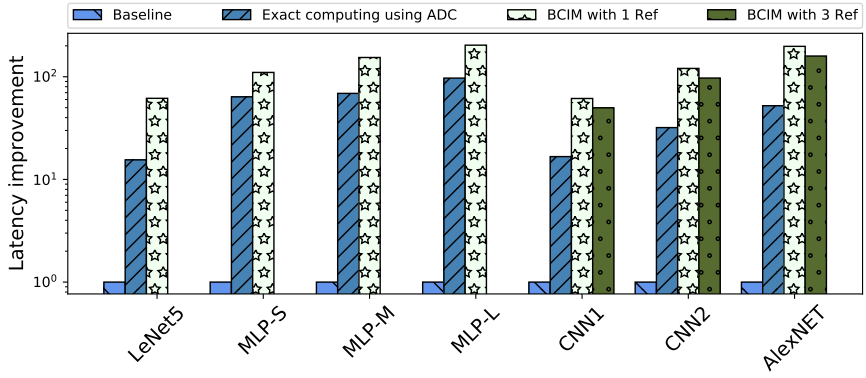


Figure 3.18: Latency improvement of BCIM compared to the baseline and exact computing design.

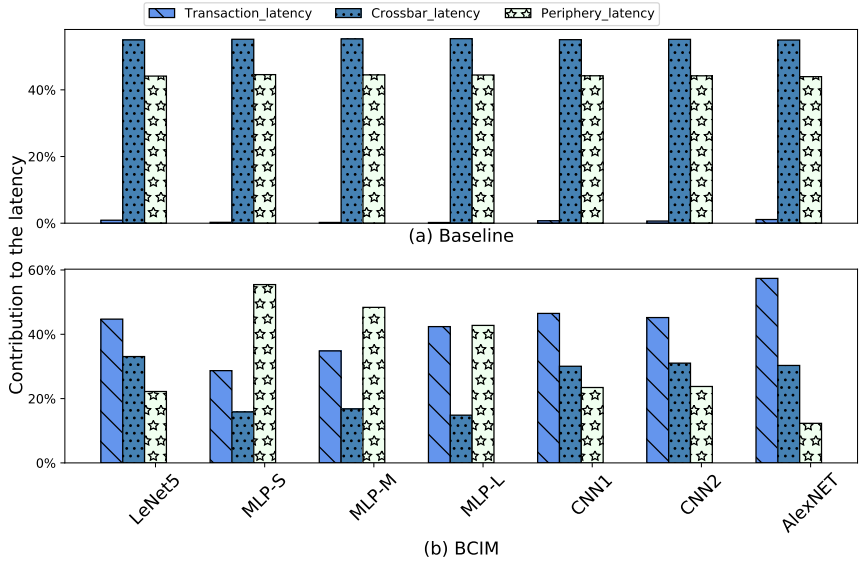


Figure 3.19: Contribution of the different parts of the design to the total latency for (a) Baseline and (b) BCIM.

values. This means more computation is performed on each crossbar with one input set than in an MLP network. Figure 3.20 (left) shows the latency breakdown per layer for AlexNet. The figure again demonstrates that the convolution layers have more contributions than fully connected layers within a network. This means in case we have to satisfy higher performance requirements in our design, more resources should be dedicated to these layers. Considering Figure 3.20(right), where we provide

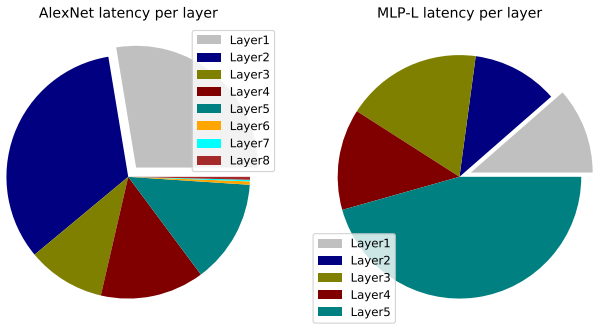


Figure 3.20: Latency breakdown for different layers of AlexNet (left) and MLP-L (right)

the breakdown of the MLP-L network, the last layer has the major contribution to the latency since this layer is not binarized, and ADCs have to be employed. Using ADC increases the contribution of periphery circuits in latency considerably. This is the reason that the periphery contributes more to the latency in our MLP networks (see Figure 3.19(b)).

3.7. LIMITATIONS

The applications of BNNs are limited since these networks provide lower accuracy (around 10% [180]) than their full precision counterparts. However, it should be noted that in this work, we avoided binarizing the entire network. Instead, the first and the last layers, which seem to have the most contributions to the accuracy loss, are kept as a non-binarized layer. This helped us to preserve the accuracy as much as possible. As we saw in Table 3.1, our binarized AlexNet network provides 80% accuracy, which is very close to the full precision network. This has to be evaluated in the future for other large networks as well. Furthermore, the approach presented in this paper can be applied to non-binary (e.g., 2-bit) weights with some assumptions. In case each memristor device can hold a weight (e.g., 2-bit) and if values are represented in the two's complement format (to be able to have both negative and positive), the same approach can be used. This can be further explored and elaborated in our future work. Hence, if a binary network suffers from huge accuracy loss, more precision weights can be potentially employed.

3.8. CONCLUSION

In this chapter, we focused on binary neural networks as a promising application for CIM. We proposed a novel in-memory memristor-based design that substantially improves both the latency and energy efficiency of BNN networks on CIM. The proposed XNOR-based BNN design replaces the ADC and digital post-processing functionality with a SA with adjusted reference(s) while maximizing parallelization and resource utilization in the design using a novel mapping of weights and activation values in the crossbar and its input buffer. The impact of SA references on

the accuracy has been evaluated at the crossbar and network levels. The design was also evaluated in terms of energy and latency for different networks and datasets. This work is able to improve energy and latency up to 5× and 100× compared to the baselines with marginal accuracy loss.

By Studying this application domain, we have learned the following:

1. CIM can potentially lead to considerable energy/performance improvement for neural networks.

2. In the case of more generic designs, flexibly supporting different datatype sizes is important. Different networks and accuracy constraints may require different datatype sizes. As we saw for BNNs, even within a network, we may have different datatype sizes (e.g., 8 bits for the first and last layer and 1 bit for intermediate layers). We discuss this more in Chapter 7.

3. Supporting multi-tiling is a must for real-world applications. A network-on-chip (NoC) is required to connect CIM-tiles and flexibly navigate data among them. Based on the NoC topology and the structure of the neural network, a mapper is required to minimize the communication overhead among the tiles.

4. Neural networks often have layers and operations that cannot be implemented in the crossbar array. For example, a pooling layer or sigmoid function should be implemented in the periphery. In the case of a more generic approach where we intend to support a wider range of neural networks or even applications, the designer should be aware of these required functionalities.

4

APPLICATION SPECIFIC DESIGN: GRAPH PROCESSING

Performing analysis on large graph datasets in an energy-efficient manner has posed a significant challenge; not only due to excessive data movements and poor locality, but also due to the non-optimal use of high sparsity of such datasets. The latter leads to a waste of resources as the computation is also performed on zero's operands, which do not contribute to the final result. In this chapter, we focus on graph processing applications and design a novel accelerator, SparseMEM, targeting sparse datasets by leveraging the computing-in-memory (CIM) concept. CIM is a promising solution to alleviate the overhead of data movement and the inherent poor locality of graph processing. The proposed solution stores the graph information in a compressed hierarchical format inside the memory and adjusts the workflow based on this new mapping. This vastly improves resource utilization, leading to higher energy and permanence efficiency. The experimental results demonstrate that SparseMEM outperforms a GPU-based platform and two state-of-the-art in-memory accelerators on speedup and energy efficiency by one and three orders of magnitude, respectively.

This chapter is based on [22].

4.1. INTRODUCTION

Graph processing is employed in a wide range of areas, including but not limited to social media analysis [143], bioinformatics [144], urban planning [145], and machine learning [146]. Graph processing is well-known for its three main characteristics [91, 147]: 1) poor locality or random access pattern to the memory, 2) simple and a small amount of computation over the accessed data, 3) high sparsity of the graphs which implies that the computations are frequently performed on zeros operands. Traditionally, the active portions of the graph are loaded sequentially into the memory hierarchy of the system while the rest of the graph is stored in the secondary memory. Due to the explosion of graphs' size, data movement across the memory hierarchy imposes a considerable overhead compared to the actual computation time/energy and limits the system's performance due to the maximum memory bandwidth. Moreover, some of this latency and energy is wasted due to the sparsity of the graph. Clearly, there is a need for new architectures and methodologies to address the challenges mentioned above.

Several hardware accelerators were proposed to enhance the energy efficiency and performance of graph processing. Some of them have focused on optimizing memory access [148], others on improving the computational efficiency [149]. To further improve the system's efficiency beyond memory bandwidth limitation, near-memory computing was deployed in some previous works; e.g., TESSERACT [150] implements a vertex-centric programming model on top of Hybrid Memory Cube (HMC). To mitigate the communication overhead between different memory cubes, numerous solutions based on efficient graph partitioning [147], configurable interconnect [151], and batched-based communication [152] were provided. To further reduce the memory bandwidth limitation, GraphR [91], GRAM [153], and GraphSAR [154] proposed promising designs by exploiting computing-in-memory (CIM) based on emerging non-volatile memristors. Unlike the works mentioned earlier, GraphSAR [154] takes into account graph sparsity and provides a CIM sparsity-aware design on top of memristor devices in which sub-graphs with low density are divided into smaller ones. This approach can partially eliminate the sparsity and has a high pre-processing overhead. Hence, there is still a need for energy-efficient solutions that minimize the data movement overhead while taking the data sparsity into consideration.

In this chapter, we propose SparseMEM, an energy-efficient design leveraging CIM; the design obtains maximum benefit from data sparsity. SparseMEM presents graph information in a hierarchical compressed format and comprises two key components: 1) Destination-Weight (DW) Crossbar, where the graph information is stored in a novel compressed representation; 2) Translation-Table (TT) Crossbar, which helps to navigate through the DW Crossbar. We implement the design using ReRAM memristor technology; memristor devices have great scalability, high density, near-zero standby power, and non-volatility [13, 21]. We compare SparseMEM with software implementation on a GPU platform as well as two CIM designs [91, 154]. The results show that we achieve 18× speed up and 2000× energy efficiency on average compared to the baselines. In short, our main contributions are:

- A novel data representation tailored for spars-based graph processing and

targeting computing-in-memory designs. This enables the computations over a compressed graph representation (stored inside the memory) irrespective of the used memory technology;

- An optimized and scalable end-to-end ReRAM-based computing-in-memory accelerator that makes use of the proposed data representation for several widely used graph algorithms. The efficiency of the accelerator is studied over the different levels of graph sparsity;
- Case studies of different workloads to evaluate the design for different performance metrics. The design is compared with a software implementation on a high-end GPU platform and two in-memory state-of-the-art designs.

This chapter is organized as follows. Section 4.2 provides background graph processing. We discuss our SparseMEM proposal in Section 4.3. Section 4.4 evaluates the design, while Section 4.5 concludes this chapter.

4.2. GRAPH FUNDAMENTALS

A widely used method of graph representation is an algebraic representation. In this method, an *adjacency matrix* is constructed where each entry (i, j) represents an edge from source vertex i to destination vertex j. This representation allows for intuitive calculations using linear algebra operations. However, naive storage of this matrix in memory does not scale. The storage occupied by a 2D matrix grows quadratically, meaning that large graphs quickly occupy an impractical amount of memory. However, by monitoring the graph information stored in the memory, we observe that most of the matrix elements do not contribute to the result of the computation since they represent an edge that does not exist between destination and source nodes. Figure 4.1 presents the sparsity of some well-known datasets [191]; this illuminates the intensity of sparsity in adjacency representation. Operating on sparse data not only increases the memory requirements, but also brings the computational efficiency down. The solution in this paper aims to maximize computational efficiency and resource utilization while performing over sparse datasets.

4.3. SparseMEM Architecture

4.3.1. OVERVIEW OF SPARSEMEM

Figure 4.2 shows the workflow of SpraseMEM compared to one of our baselines GraphR [91]. As stated before, real-world graph datasets are extremely large, even using compressed representations. However, the size of the storage unit, ReRAM memory, is practically limited due to the current technology restrictions. Therefore, we need to store the entire preprocessed graph dataset on disk. In the GraphR approach (as shown in Figure 4.2), the computation and storage are distinguished even within the ReRAM crossbars. While the first part (ReRAM Memory) holds the graph information loaded from the disk, the second part (ReRAM Graph engines)

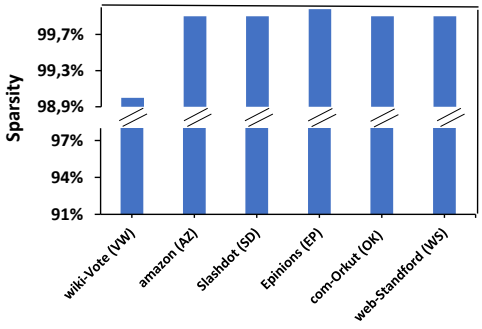


Figure 4.1: Percentage of sparsity in adjacency matrix over some well-known graph datasets.

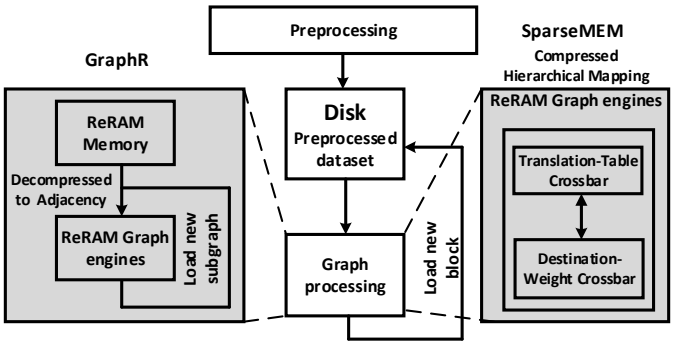


Figure 4.2: SparseMEM and GraphR workflow comparison. Reducing the overhead of device programming by enabling computation over a compressed representation.

performs the computation over the uncompressed information. Nevertheless, the following challenges are faced: (a) Considerable data movement between ReRAM memory and Graph engines reduces the performance and energy efficiency of the system; (b) Conversion from the coordinate list (a compressed representation) to the adjacency representation (used for processing) imposes extra processing overhead on ReRAM engines in each iteration; and (c) Mapping the adjacency matrix to the graph engines leads to poor resource utilization due to high data sparsity. It is worth mentioning that increasing the number of memristor device programming reduces the endurance and energy efficiency.

SparseMEM presents the graph information in a new compressed *hierarchical* format inside the ReRAM crossbars. The design comprises two main components: 1) Destination-Weight (DW) Crossbar, where the graph information is stored in compressed representation 2) Translation-Table (TT) Crossbar, which decodes the information in the DW Crossbar and guides to extract information regarding the positioning of vertices which is needed for computation. This allows us to perform

the computation exactly where the data is stored inside the ReRAM memory. Hence, there is no separation between ReRAM memory and the ReRAM processing unit. The SparseMEM major differentiators are: (a) it alleviates the number of data loading from disk due to the efficient use of ReRAM memory; (b) it uses computational resources efficiently by performing computation over compressed information; and (c) it eliminates the data conversion from compressed to adjacency representation.

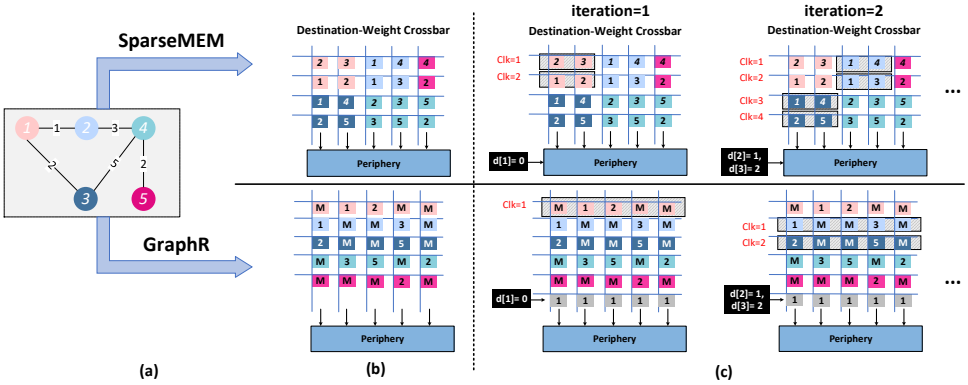


Figure 4.3: (a) Graph example; (b) mapping of graph information into crossbar for computation phase considering SparseMEM and GraphR design; (c) the first two iterations of the SSSP algorithm starting with source vertex 1.“M” means no connection

4.3.2. Graph Mapping and Data Representation

In this section, we explain our compressed hierarchical representation by providing a simple example based on a toy graph depicted in Figure 4.3(a). To clarify more on SparseMEM and identify the differences, we compare it with the GraphR design [91], where the adjacency matrix is directly mapped to the crossbar (similar to GraphSAR [154]).

In the case of GraphR (below part of Figure 4.3), where the adjacency matrix is directly mapped to the crossbar, the storage of edge weights encodes more information than just the weight. Since the entries are expanded, the location of the edge in the crossbar also encodes the source and destination vertex of this edge. Figure 4.3(b) shows the mapping used in GraphR. As an example, the *first* row of the crossbar shows which vertices of the graph are connected to vertex 1, the *second* row in the crossbar gives the vertices connected to vertex 2, etc. For example, vertex 1 is connected to vertices 2 and 3; entities in the matrix give the weight of connections (1 and 2) and “M” denotes no connection. However, this information is lost when the edges are stored in a compressed format. Thus, a proper mapping to preserve this information is required.

In the case of SparseMEM design (top part of Figure 4.3), optimal use of storage

is made. Each graph's vertex gets a *sub-array* in the **Destination-Weight (DW) Crossbar**; each of these sub-arrays consists of two rows: one for the index of *destination* vertices and one for the *weights* of the edges connecting the source to destination vertices; the number of columns in a sub-array depends on the connectivity of a vertex to other vertices. In Figure 4.3(b), each color in the DW Crossbar represents a sub-array for a vertex. For example, the pink color presents the sub-array associated with vertex 1. As vertices 2 and 3 are connected to vertex 1, we store the index of these vertices in the *first* row (being 2 and 3) and their weights (being 1 and 2 respectively) in the *second* row. Note that, in SparseMEM, only the non-sparse data is stored, which is required for computation. E.g., for vertex 1, only four values are stored in DW Crossbar. However, in the GraphR design, the entire row 1, which represents the collection of edges with source vertex 1, has to be stored. All these devices contribute to the execution, even though only two hold the data of interest, and the rest hold a predefined value representing no connection.

The key question is now how to preserve the information regarding the location of edges belonging to a vertex in the DW Crossbar. In order to encode this information, a separate **Translation Table (TT) Crossbar** is used. This crossbar is employed to encode the location of the edges of a particular source vertex (being stored in the DW Crossbar) and navigates through it. In the TT Crossbar, we store the information for each vertex as 'start address' and 'end address'; these refer to the first and last location, respectively, occupied by a vertex in the DW Crossbar. Figure 4.4 illustrates an example of 4×4 TT Crossbar; it assumes the addressing is performed in an increment manner from left to right (i.e., fast column addressing). The first two addresses are reserved for vertex 1, the second two addresses for vertex 2, etc. For example, the start and end addresses of vertex 3 stored in the TT Crossbar are 6 and 7, respectively. To translate the address to the DW Crossbar addresses, the following formula is used: $A_{DW} = A_{TT} + \lfloor A_{TT}/C \rfloor \times C$ where C denotes the number of columns in the DW Crossbar and $\lfloor\rfloor$ denotes the floor of division. In the example of Figure 4.4, C=5; hence, for vertex 3, A_{DW} is 11 (start) and 12 (end), assuming also fast column addressing of DW Crossbar. Note that, the translation of A_{TT} to A_{DW} skips the even rows as they always store the weights corresponding to A_{DW} addresses (vertices). This hierarchical storage format eliminates sparsity, while still preserving edge source and destination. TT Crossbar is operating in parallel with DW Crossbar and can provide information to multiple DW Crossbars regarding the address of active vertices for the next iteration. Several DW Crossbars together with their TT Crossbar form a cluster. A design may contain several clusters.

4.3.3. EXECUTION FLOW

In this subsection, we explain the execution flow of SpraseMEM by providing an example based on Single-Source Shortest-Path (SSSP) algorithm (see Algorithm 1). In graph algorithms, the execution can be divided into two steps: a) compute and b) update. Considering the SSSP algorithm, we take a single start vertex and compute the distances "d" to every other vertex in the graph. The vertices whose distance values are updated in the current iteration are activated for the next iteration. The algorithm continues until there are no active vertices. When we access the edge that

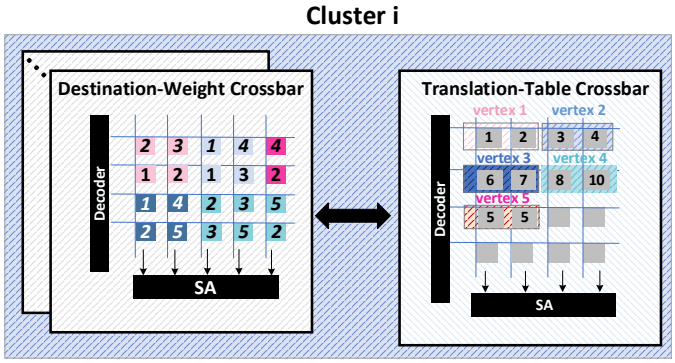


Figure 4.4: Mapping of information regarding positioning of vertices in the DW Crossbar into the TT Crossbar

connects an active source vertex to a destination vertex, the edge weight has to be added to the current distance value of the source vertex (compute step). At the end, we need to update the destination registers (update step) where we store the distance value of vertices to the source vertex. This is usually a simple mathematical operator and varies across algorithms. In the case of the SSSP algorithm, this operator is a *Min* function that gives the minimum of the current computed distance value as well as the old value for a vertex. Next, we illustrate the algorithm for SparseMEM and GraphR implementation.

GraphR: Figure 4.3(c) illustrates the first two iterations (line 3 in Algorithm 1) of the SSSP algorithm where we want to find out the distance of vertex 1 to other vertices. To clarify more on the implementation, we consider the second iteration of the algorithm in Figure 4.3(c) as an example where we want to find the distance from vertex 1 to the rest of the vertices through vertex 2 and 3 (they got activated after the first iteration). When we get access to the edges belonging to vertex 2 (i.e., $weight_2[4] = 3$), the current distance from vertex 1 to vertex 2 ($d[2]$), which was computed in the first iteration, has to be added up ($New\_d[4] = d[2] + weight_2(4)$). In the GraphR implementation, this addition is performed in an analog manner where the current distance of the active vertex $d[2]$ is given to the crossbar as input, as shown in the bottom part of Figure 4.3(c). Note that the last row is programmed to value 1. This is because $d[2]$ has to be first multiplied by 1 and then added to the second row storing the edge weights belonging to vertex 2 (e.g., $weight_2[4]$). Due to the limited resolution of the input drivers, input data has to be sliced and applied to the crossbar in several steps. As a consequence, the implementation of such addition requires costly peripheral components like power-hungry ADC and Shift-and-Add units, as shown in Figure 4.5(a). After the compute step, the update step takes place in the periphery of the DW Crossbar using a digital circuit (Min in Figure 4.5), and stores the result in the destination register dedicated to this vertex.

SparseMEM: To demonstrate how SparseMEM performs computing, we consider the same case as we did for GraphR; i.e., the second iteration of Algorithm 1

Algorithm 1 SSSP algorithm

```
1: ActiveVertices[start] ← True
2: d[start] ← 0
3: while ActiveVertices ≠ ∅ do
4:     for v ∈ ActiveVertices do
5:         for each neighbour u of v do
6:             New_d[u] ← d[v] + weight_(v)(u)  // compute
7:             d[u] ← min(New_d[u], d[u])  // update
8:             if d[u] changed then
9:                 ActiveVertices[u] ← True
10:            end if
11:        end for
12:    end for
13: end while
```

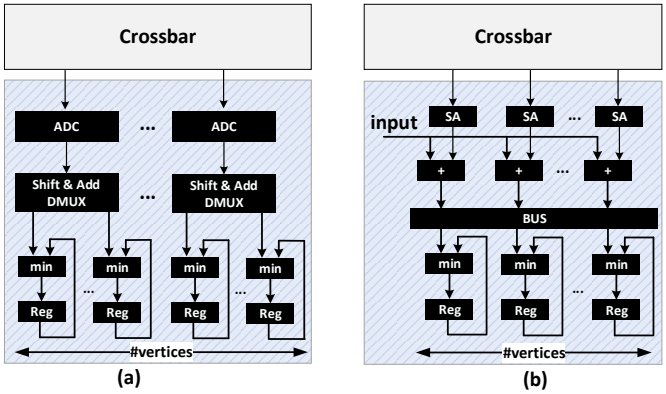



Figure 4.5: Periphery design for (a) GraphR and (b) SparseMEM

where we aim at finding the distance from vertex 1 to vertex 4 through vertex 2. The compute step (i.e., $New\_d[4] = d[2] + weight_2(4)$) is calculated by first reading $weight_2(4)$ from DW Crossbar (see iteration 2 in Figure 4.3) and then adding it to $d[2]$, which is provided as input to the periphery of the crossbar. The addition is done with a digital adder as shown in Figure 4.5(b). This approach avoids activating the crossbar several times and using Shift-and-Add units. This also helps to replace expensive ADCs with simple Sense Amplifiers (SAs). However, SparseMEM needs a bus or an interconnect component. Due to the compressed representation in SparseMEM, the crossbar no longer encodes vertex location. Therefore, a bus is placed in order to navigate data to the target destination register. As already mentioned, before computing the new distance value (e.g., $New\_d[4]$), the index of the vertex where this value belongs (e.g., 4) is read from the DW crossbar. This information is used to configure the bus and navigate the new distance value to its destination register.

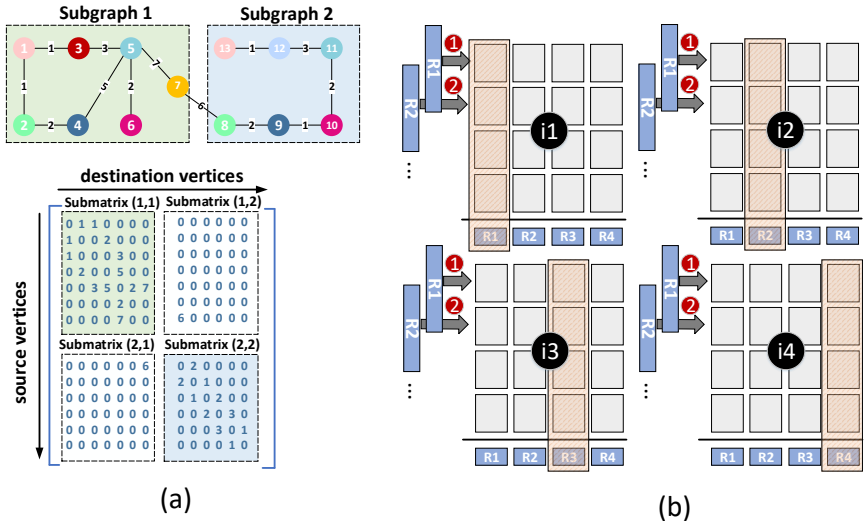


Figure 4.6: (a) Partitioning the adjacency matrix into sub-matrices (b) column-major execution model for streaming the sub-matrices (each square is a sub-matrix)

4.3.4. Sub-graphs streaming and processing

To process graphs much larger than the available memory size provided by the crossbars, it is necessary to stream the graph data into the crossbar from another source, such as secondary memory storage. This is considered for GraphR, GraphSAR, and SparseMEM. To stream a graph, we split it up into "sub-graphs". To be more specific, the adjacency matrix is partitioned into sub-matrices which either represent the connectivity within a sub-graph or between two sub-graphs. Figure 4.6(a) shows an example of graph partitioning. In the case of GraphSAR [154] design, sub-matrices in which all elements are equal to zero are eliminated from the process to improve efficiency in the presence of high data sparsity. In SparseMEM, while the sub-matrices are streamed from the disk to the DW Crossbars, we program the TT Crossbar as well. This information is known at compile time and does not require extra processing during the execution. Such a format of graph streaming allows for processing graphs much larger than the available memory size provided by the crossbars.

Figure 4.6(b) shows the processing order of sub-matrices. There are two main approaches: 1) column-major and 2) row-major. Selecting the right approach may enhance the performance and energy efficiency of the design. In column-major (row-major) order, sub-matrices with the same destination (source) vertices are processed in each iteration. We indicate the iterations in Figure 4.6(b) with black circles. Considering row-major order, the content of all destination registers has to be loaded from the disk into the crossbar for each iteration. Similarly, in column-major order, the content of source registers has to be loaded together with

the corresponding sub-matrices in sequence (red circles) to obtain new distance values. However, in column-major order, we only need to load the current distance to the "active source vertices" which are usually less than the total number of source vertices. Therefore, there is no need to load all the source registers. Hence, it leads to less communication to the disk during the execution. It should be noted that, due to the current technological limitations, limited crossbars can be employed in realistic design. Hopefully, as the limitations fade away in the future, memristor crossbars can be replaced with the secondary memory storage; hence there would be no/less concern regarding the communication overhead between these two memories.

4.4. EVALUATION AND DISCUSSION

4.4.1. EXPERIMENTAL SETUP

Our simulation results are obtained by creating a platform for SparseMEM written in C++ [181], which takes graph datasets and performs the same steps as the hardware. These steps include the various operations performed on the crossbar, such as reading and writing to the crossbar. The parameters for ReRAM technology are taken from [192]. We assume each memristor cell can hold one bit (two resistance levels) for all the simulations. The parameters for ADCs are taken from [193] given in 32 nm technology, and the resolution of the input drivers for the crossbars are 1-bit [194]. We summarize the parameters regarding the crossbar technology in Table 4.1. Digital peripheries are synthesized in Cadence Genus targeting standard cell 15 nm Nangate library. Finally, we use 16-bit integer data size in all experiments.

Table 4.1: Memristor tile specification

Crossbar - ReRAM (128x128 @1bit)		
	Energy (Single Cell)	Latency
Read	40fJ	10ns
Write	20pJ	100ns
ADC (8-bit)		
Energy	2pJ per sample	
Latency	1ns per sample	
Shared with	32 columns	
SA		
Energy	0.01pJ per sample	
Latency	1ns per sample	
Shared with	4 columns	

We evaluate SparseMEM in terms of speedup and energy while comparing the solution with GraphR [91], GraphSAR [154], and Nvidia Geforce RTX2080 GPU platform. We use Nvidia-smi to obtain the power consumption of the GPU platform. Our experiments concern three algorithms (application): 1) SSSP, 2) Breadth First Search (BFS), and 3) PageRank on real-world graph data sets, which are retrieved from the SNAP graph repository [191]. Datasets (workload) used for the experiments

Table 4.2: List of graph datasets

Dataset	Average Degree	#Vertices	#Edges	Domain
wiki-Vote (WV)	29	7k	104k	Social
amazon0302 (AZ)	9	262k	1.23M	Co-purchasing
soc-Slashdot0902 (SD)	23	82k	948k	Social
soc-Epinions1 (EP)	13	75k	508k	Social
com-Orkut (OK)	39	3M	117M	Communities
web-Stanford (WS)	16	281k	2.3M	Web

are summarized in Table 4.2.

4.4.2. EXPERIMENTAL RESULTS

Figures 4.7,4.8, and 4.9 depict speedup and energy improvements of SparseMEM w.r.t the baselines for SSSP, BFS, and PageRank algorithms, respectively. It shows the following:

1. Speedup is strongly application and workload-dependent. E.g., SparsMEM archives minimum speedup for PageRank, and minimum speedup for 'WV' workload.

2. SparseMEM systematically outperforms in terms of energy efficiency irrespective of the workload and application.

3. On the average, SparseMEM achieves 18× speedup and 2000× energy improvement compared to CIM baselines.

Speedup: Sparsity is the main factor determining the speedup in SparseMEM compared to the baselines. Figure 4.10 depicts the utilization of memristor devices in SparseMEM and GraphR implementations; this represents the total number of (re)programming memristor devices during the execution of two algorithms. Less reprogramming devices in SparseMEM leads to significant improvement in speedup. According to the results, the minimum speedup improvement is for the Wiki-Vote (WV) workload with the lowest sparsity (see Figure 4.1); as sparsity reduces, the overhead of hierarchical mapping used in SparseMEM increases. Note that even if graphs have similar sparsity (e.g., 'AZ' and 'SD' workloads in Figure 4.1), the speedup can be different (see Figures 4.7,4.8, and 4.9); this is due to the distribution of data over sub-graphs (graph connectivity). If the data is scattered over many sub-graphs, there would be fewer sub-graphs whose all elements are zero. Consequently, there are fewer sub-graphs to be eliminated from the process, which increases the overhead in the baselines. Among applications, SparseMEM achieves similar (or less) performance to the baselines for the PageRank algorithm. Since the baselines directly map the adjacency matrix to the crossbar, analog matrix multiplication can be supported inside the crossbar. Therefore, they can achieve more parallelization for PageRank, where matrix multiplication is an essential kernel.

Energy: Sparsity and the periphery design are also major factors influencing energy consumption. As stated in sub-section 4.3.3, the input drivers and ADCs

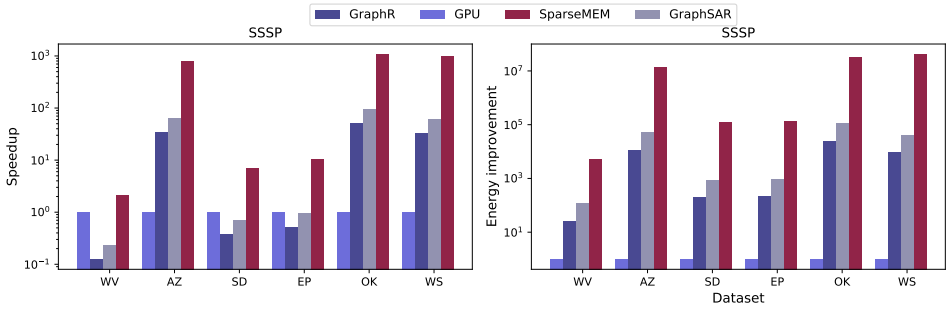


Figure 4.7: Speedup and energy improvement of SparseMEM compared to the baselines for SSSP algorithm (normalized to GPU)

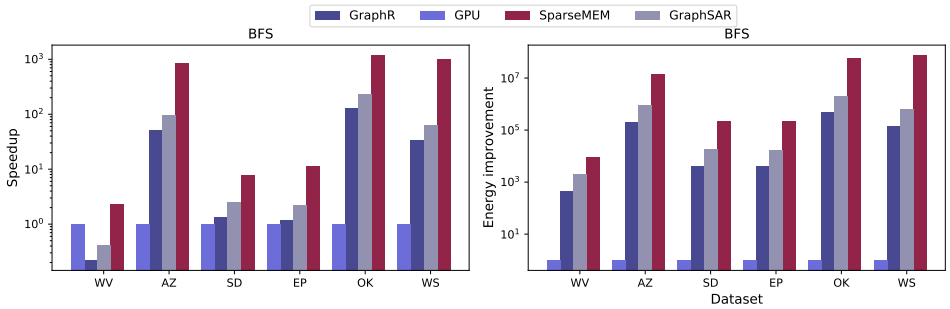


Figure 4.8: Speedup and energy improvement of SparseMEM compared to the baselines for BFS algorithm (normalized to GPU)

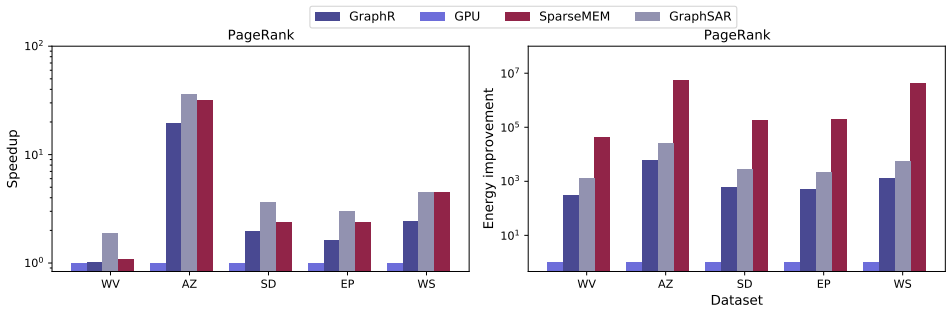


Figure 4.9: Speedup and energy improvement of SparseMEM compared to the baselines for Pagerank algorithm (normalized to GPU)

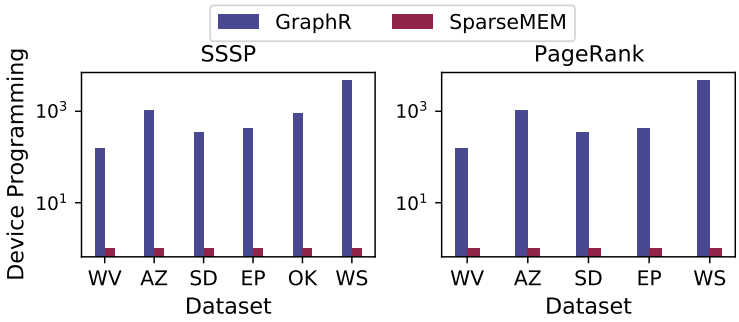


Figure 4.10: Relative number of memristor programmings required by SparseMEM and GraphR (normalized to SparseMEM)

are major energy consumers. In GraphR implementation, feeding a data operand (e.g., 16 bits size) to the input driver requires several (e.g., 16) iterations due to the limited driver resolution [26]. This increases both energy and latency. In addition, using ADC in the periphery consumes more energy. Hence, higher energy overhead is imposed on the system. SparseMEM, however, is a read-driven design where the computation (e.g., addition) is performed in the periphery. This reduces the energy required for computation in the crossbar as well as its periphery.

4.5. CONCLUSION

This chapter proposes SparseMEM, an in-memory graph processing accelerator tailored for sparse workloads. The key idea is the compressed and hierarchical mapping of the graph information into memristor-based crossbars to efficiently perform the computation inside the memory. The design requires less reprogramming of memristor devices (i.e., read-driven), while performing data-centric computing (CIM). The result implies the importance of hardware/mapping co-design for energy-efficient CIM design.

5

APPLICATION SPECIFIC DESIGN: BIOINFORMATICS

Taha SHAHROODI and Mahdi ZAHEDI

The knowledge of bioinformatics is used in a wide range of applications. Food industries, as high-risk enterprises that are directly associated with human health, are leveraging this knowledge in their food monitoring systems to increase the quality and nutritional value of foods. Food profiling is an essential step in any food monitoring system needed to prevent health risks and potential fraud in the food industry. Food profilers work on massive data structures and incur considerable data movement for a real-time monitoring system. To this end, we propose Demeter, the first platform-independent framework for food profiling. Demeter overcomes the first limitation through the use of hyperdimensional computing (HDC) and efficiently performs the accurate few-species classification required in food profiling. We overcome the second limitation by the use of an in-memory hardware accelerator for Demeter (named Demeter) based on memristor devices. Demeter actualizes several domain-specific optimizations and exploits the inherent characteristics of memristors to improve the overall performance and energy consumption of Demeter. We compare Demeter's accuracy with other industrial food profilers using detailed software modeling. We synthesize Demeter's required hardware using UMC's 65nm library by considering an accurate PCM model based on silicon-based prototypes. Our evaluations demonstrate that Demeter achieves a (1) throughput improvement of 192× and 724× and (2) memory reduction of 36× and 33× compared to Kraken2 and MetaCache (2 state-of-the-art profilers), respectively, on typical food-related databases. Demeter maintains an acceptable profiling accuracy (within 2% of existing tools) and incurs a very low area overhead.

This chapter is based on [23].

5.1. INTRODUCTION

This chapter focuses on food profiling as the third and last application in this thesis. Food profiling is the first step and the only computationally expensive task in a food monitoring system. The food profiling task entails the functionality of determining the existing species in a food sample and their relative abundance [195, 196]. Today's food profilers work with sequenced data as we can capture a more accurate profile using the sequences of a food sample. The rapid drop in the cost of DNA sequencing in the past decades and the expectation for a continual trend [197, 198] is expected to lead the way for profiling to become the main bottleneck of this pipeline.

We pinpoint two critical sources of inefficiency in state-of-the-art (SOTA) profilers currently used for food monitoring, collectively called food profilers or profilers hereafter. First, all current (food) profilers work with significantly large working data structures, e.g., humongous hash tables or sorted lists, that require high-end servers with extensive storage and memory capabilities to be handled. Second, current profiling techniques incur a significant number of random accesses to large working datasets, and as a result, unnecessary data movement between their storage and memory plus their memory and compute units which cannot be otherwise done where the data resides due to (1) the size of the final data structures and (2) the required operations for tasks in hand. This directly translates to massive energy consumption and latency. For example, as shown in our evaluations, a widely used SOTA profiler takes ~1 minute to profile one high-coverage sequenced food sample. However, it requires a super machine or cluster with at least 300 GB of memory and proportionally scaled-up compute power. Therefore, a healthy economy regarding the food industry cannot keep using these profilers and demands cheaper, faster, more energy-efficient, and more accurate food profilers for the years to come.

Our goal in this chapter is to solve both limitations of previous profilers, namely (1) reliance on high-end servers and scaling problem due to required massive data structures and (2) incurring unnecessary data movement. **To this end,** we propose Demeter, an CIM-based hardware/software co-designed food profiling accelerator that efficiently profiles species of a food sample. **The key idea** of Demeter is to reduce the food profiling problem to a multi-object (multi-species) classification problem using hyperdimensional computing (HDC) followed by an abundance estimation step. We provide the necessary background information about HDC in the following section.

This chapter makes the following main contributions:

- The first framework that enables food profiling via HDC. Our CIM-enabled hardware accelerator uses memristor devices (Demeter) to extract Demeter's full potentials and solve the data movement problem. We propose several optimization techniques for Demeter based on domain-specific knowledge of food profiling and our background on PCM cell characteristics and HDC operations. To our knowledge, Demeter is the first (in-memory) hardware accelerator for a food profiler (Section 5.4).

- We rigorously compared Demeter and Demeter to four SOTA food profilers. We show that Demeter provides an accuracy level comparable with previous food

profilers and within the accepted level of food monitoring systems. The default setting of Demeter enables a (1) throughput improvement of ~192× and 724× and (2) reduction in the required memory of ~36× and 33× compared to Kraken2 [199] and MetaCache [195], respectively, when querying on a typical food-related reference genome database, i.e., AFS20 [195]. Our design requires only ~8.9 mm^2 die area and can process ~9.45 Mbp per joule for our largest food-related database AFS31 [195] (Section 5.5).

5.2. BACKGROUND

This section discusses the necessary background on HDC. For more detailed background information, we refer the reader to the original paper [23] and comprehensive reviews on other required topics [200–203].

5.2.1. HYPERDIMENSIONAL COMPUTING

Hyperdimensional computing (HDC) [204, 205] is a brain-inspired computing paradigm that has been demonstrated to be effective in reference-based learning domains, such as text classification [206], gesture recognition [207], and latent semantic analysis [208]. Elements of HDC are presented using high-dimensional vectors, hereafter called HD vectors. HD vectors can be composed of real, binary, bipolar, or complex numbers. Previous works show that binary representations of HD vectors are more practical and efficient for classification problems or one-shot reinforcement learning. This representation is also more hardware-friendly. Therefore, we proceed with binary HD representations. HD vectors also come with other powerful features, such as robustness to random errors, holistic representation, and randomness. We refer the enthusiastic readers to previous works for more details on these features [204, 209]. In addition, some theoretical background on HDC can be found in [210]. Finally, a complete overview of recent works on HDC, their implementation, and applications is presented in [211].

Like other reference-based classifiers, an HDC-based system also takes two steps: (1) training and (2) classification. An encoding mechanism is used in both steps. One famous example is the N-gram encoding mechanism that follows a two-step approach for encoding a string of size L to an HD vector of size D. **Step 1:** It combines N consecutive alphabets of the string and builds an HD vector that is orthogonal to them all and can preserve their relative order. This operation is called binding and is represented in Equation 5.1, where $\rho^i(X)$ represents the i^{th} permutation of vector X and B_i are once randomly-generated representative HD vectors (also referred to as atomic or basis HD vectors) for the i^{th} character of the string C_i. In the equation, "Sh" is a shift operation. The string is a DNA sequence in Demeter.

$$N-gram(C_1, C_2, ..., C_N) = Sh[...Sh[Sh[B_1] \oplus \rho(B_2)] \oplus ...] \oplus \rho^{N-1}(B_N) \quad (5.1)$$

Step 2: The encoder performs an element-wise addition between all HD vectors corresponding to consecutive N-gram, called bundling, to present the entire input

sequence. To binarize the final HD vector, the encoder applies a majority function over each position. This final vector is stored in associate memory (AM) and is called a prototype HD vector if the input was a reference genome. Otherwise, it is called query HD vector and will use it for classification.

The most common approach for classifying whether the sequence query belongs to any of the classes in AM after using N-gram encoding mechanism is to measure the hamming distance between the query HD vector (Q) and each of the prototype HD vectors (Ps) and decide based on a fixed distance or threshold (T). This can be easily performed with an XNOR of Q and each P followed by a pop-count and thresholding operation, as shown in Equation 5.2.

$$Classification(i) = \begin{cases} 1, & \text{if } \sum_{j=1}^{D} Q(j) \bar{\oplus} P_i(j) \geq T \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

5.3. FRAMEWORK

Although the current food profilers are accurate, they are neither memory- nor energy-efficient. The primary sources of high cost and inefficiency in current food profilers are their large reference data structures and, consequently, the significant overhead of data movement. To address this challenge, we propose a framework for food profiling, Demeter, in this chapter. The framework stands at the intersection of HDC, CIM, and bioinformatics. Figure 5.1 provides an overview of the five key steps in Demeter: ❶ defining an HD space, ❷ building an HD reference database (HD-RefDB), ❸ converting sample reads into HD space, ❹ determining the possible species assignment per sample read, and ❺ performing abundance estimation. We describe each step in more detail next. The hardware implementation of the accelerator is provided in the next section.

5.3.1. STEP 1: DEFINE THE HD SPACE

As the first step, Demeter defines an HD space for all subsequent operations and steps. This is a crucial step as it determines the operations in the remaining steps. Unfortunately, many previous HDC-based proposals did not support the user's input for determining the HD space and designed their space statically. Hence, such designs are more limited.

We define the HD space in 4 stages. **Stage 1:** We fixe two hyperparameters: (1) The dimension of the HD space; i.e., the dimensionality of the HD vectors (element representations), and (2) The sparsity of each element (HD vector). **Stage 2:** We generate a few atomic HD vectors and store them in memory (commonly called Item Memory (IM)). These vectors can be (1) the HD vectors that represent our genome alphabets or (2) the one-time randomly generated HD vectors that some encoding mechanisms use, for example, to introduce the concept of order between alphabets of one input. **Stage 3:** We decide on the encoding mechanism to build the space with. This encoding mechanism will be used throughout Steps ❷ and ❸ of Demeter. **Stage 4:** We fix the similarity metric and any other associated parameters (such

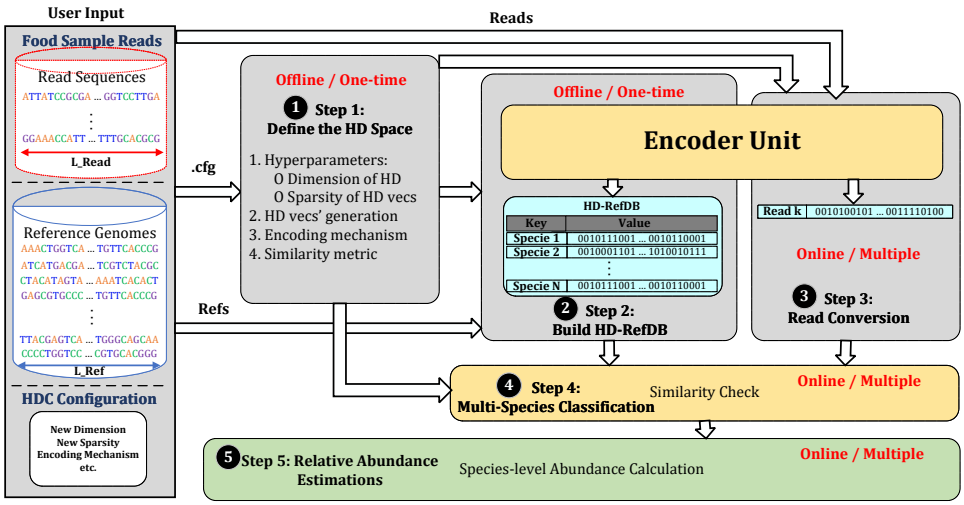


Figure 5.1: Overview of Demeter framework.

as thresholds) based on the user's input or a common choice considering previous stages.

5.3.2. STEP 2: BUILD DEMETER'S REFERENCE DATA STRUCTURE

We have two sets of inputs in step ❷: (1) HD space parameters defined in the first step and (2) a reference genome database. Subsequently, we build a new reference database in HD space out of all the considered reference genomes. This new database called HD-RefDB, consists of one (or few) prototype HD vector(s) from any given reference genome in the original reference database and is stored in AM. This step has to be done only once for the reference data. Hence, we perform this offline.

5.3.3. STEP 3: DEMETER'S READ CONVERSION

Demeter takes two inputs in Step ❸: (1) HD space configuration and (2) read sequences of the food sample under study. Demeter translates each of these read sequences into one query HD vector. To prevent any extra storage cost and to pipeline computations of the third and fourth step, Demeter forwards each query HD vector to the next step instead of storing them inside a memory unit while waiting for all of them to be constructed first. Query HD vectors created in this step can require larger or smaller space than a read, depending on the initial length of the read sequences and the dimension of the HD space. Therefore, although Steps ❷ and ❸ share the encoding mechanism, their input and how Demeter treats the outcome are pretty different. Step ❸ neither introduces a new operation nor a procedure other than those that already exist from the second step. Therefore, it enjoys similar

benefits as the second step, namely high parallelization and in-memory suitability. Demeter runs Step ❸ every time it profiles a new read of a food sample.

5.3.4. STEP 4: MULTI-SPECIES CLASSIFICATION PER READ

In this step, Demeter takes (1) the query HD vector (Step ❸), (2) prototype HD vectors from the reference database (Step ❷), and (3) similarity function and its corresponding parameters (Step ❶) as inputs. To determine the specie(s) that each read belongs to, Demeter performs a similarity check between the query HD vector and each of the prototype HD vectors. Usually, this step can be implemented only with simple operations. It also enjoys high parallelization, similar to the previous steps.

Demeter may find out that the query HD vector is close to one, multiple, or none of the prototype HD vectors in the database. This variety in possible outputs differentiates Demeter from many previous HDC-based designs [63, 190]. In such works, mostly due to the characteristics of applications under study, researchers always assume that (1) the query HD vector can only belong to one of the prototype HD vectors, and (2) the class of the query HD vector will exist in the AM. However, none of these assumptions hold for a food profiler. One read from the food sample can be related to one, multiple, or none of the reference genomes in the original reference genome database. This is because the read sequences are mostly short strings with a reasonably high probability of existence in longer reference genome sequences. It is also not uncommon that the query HD vector does not belong to any of the reference genomes in the initial reference genome database. This case can happen when, for example, (1) there is either an unknown species in the food sample, (2) one incorrectly excludes the corresponding reference genome in the initial reference genome database, or (3) an uncorrected sequencing error has happened. This difference between how many prototype HD vectors in the database can be assigned to one query HD vector is a key difference that affects both the following abundance estimation step and final results. It also distinguishes this work further from previous HDC-based proposals for different applications. Step ❹ also enjoys high parallelization and in-memory suitability features similar to previous steps.

5.3.5. STEP 5: SPECIES LEVEL ABUNDANCE ESTIMATION

In Step ❺, we need to perform a relative abundance estimation based on the results of the fourth step. This step is particularly needed for a food profiler where one query HD vector can be similar to one or more classes/species. We categorize each query HD vector into (1) uniquely-mapped, (2) multi-mapped, and (3) unmapped. Step ❺ can be extended to support different assignment policies for the multi-mapped reads. We leave investigating the effect of such methods for future work.

5.4. DEMETER HARDWARE IMPLEMENTATION

We focus on the hardware implementation of Demeter in this section.

5.4.1. OVERVIEW OF DEMETER'S ACCELERATOR

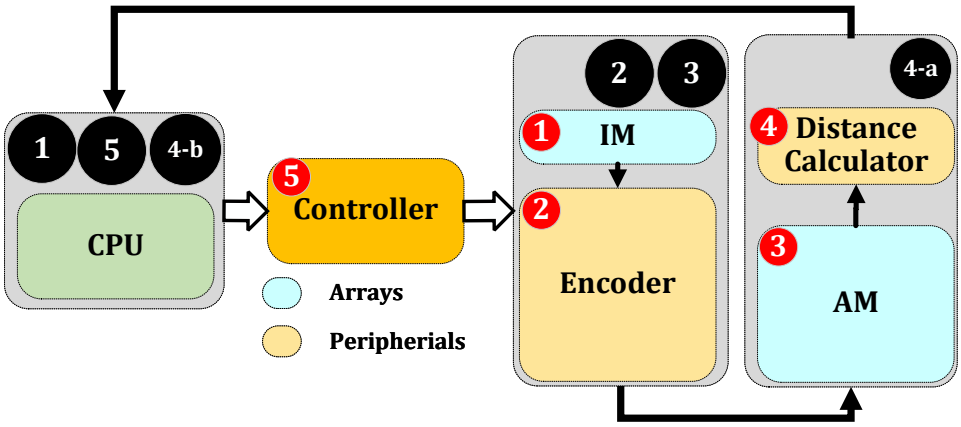


Figure 5.2: Overview of Demeter's in-memory accelerator.

Figure 5.2 shows an overview of the proposed CIM-enabled hardware accelerator for Demeter. The hardware consists of 5 key elements: ❶ Item Memory (IM), ❷ Encoder, ❸ Associate Memory (AM), ❹ Distance calculator, and ❺ Controller. IM and AM units are memory units, and we implement them as PCM arrays with their control circuitry. However, the encoder and distance calculator units are computing units implemented as the periphery. The controller is a simple FSM designed to harmonize the required steps of Demeter. The CPU initiates Demeter by gathering the user's input and then booting the controller; i.e., it sends the start command, initializes the registers, and sets the addresses to consider for food samples and/or reference genomes in the controller. The controller returns the results of the distance calculator to the CPU for final processing and performing the relative abundance estimation. We discuss these units in more detail next.

5.4.2. ITEM MEMORY (IM) DESIGN

We implement our IM using PCM arrays and corresponding circuits, such as decoders. IM stores the atomic HD vectors. Binary "0" and binary "1" in an HD vector translate to amorphous and crystalline states, respectively. Initially, the user (or Demeter) generates HD vectors for each DNA alphabet and its every permutation in an N-gram in Step ❶ of Demeter and stores them in the IM. Demeter reads these atomic HD vectors from IM every time it meets a new symbol. Once Demeter fixes the HD space, IM becomes a read-only memory. This allows us to prevent unwanted changes to the atomic vectors.

Figure 5.3-(A) presents the IM design. The gate enabler provides access to cells that the row decoder activated. This way, the design of an entire array is achieved much easier, and the write/read disturbance effect is also mitigated to a great extent. However, this design also blocks the write on a row basis and only allows column-wise programming of IM. This does not complicate IM in any way because the atomic vectors are generated once in the beginning by the host CPU and then stored in the IM for a long time. Note that random number generators are already well-optimized in CPUs. In addition, randomly generated values inside memristors are still in the early stages, and Demeter can be modified later to benefit from a non-intrusive (compatible) random number generator in the future.

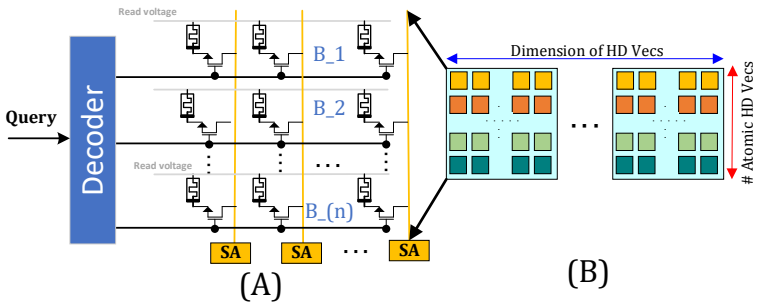


Figure 5.3: (A) IM design. (B) Data Mapping and placement of atomic HD vectors in IM.

Figure 5.3-(B) presents (1) data mapping and (2) placement of HD vectors in the IM unit. IM enjoys row-major data mapping. A row-major data mapping of HD vectors allows Demeter to read the cells written in one row in one cycle. This is helpful as IM is used in the encoding procedure, which is the bottleneck. An important design choice regarding IM is related to the limited size of PCM arrays (512×2048 [190]). This limitation of array size prevents us from fitting an entire large HD vector in one row or column. Therefore, one needs to break such an HD vector into smaller chunks and store them in separate rows. Three options exist: (1) putting the chunks in the same array, (2) putting them in different arrays, (3) a hybrid approach. To prevent exacerbating the overhead of the encoding procedure, as the encoder is already the bottleneck of the system, IM breaks a HD vector to the largest power of two that is smaller than the number of columns available in an array and stores different chunks on different arrays. This is a direct tradeoff between the used area (#arrays) and performance.

5.4.3. ENCODER DESIGN

The encoder is the main compute unit of Demeter. The encoder is implemented in the periphery of arrays and executes the binding and bundling operations via a sequence of commands determined by the controller. In this setup, the N-gram encoding mechanism is the most common one, which Demeter supports. We

suspect that other choices are also possible with the same hardware or minimal changes. We leave the exploration of those designs for future work.

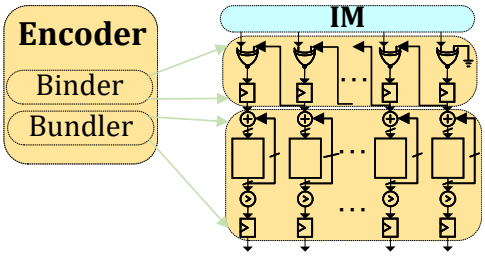


Figure 5.4: Encoder components and schematic.

Based on Equation 5.1, building an N-gram requires only simple XOR and shift operations. This bitwise XOR operation can be computed in the periphery. For the binding, the SA reads out the value from IM to one input of an XOR gate and uses the previously stored value of neighbor FFs as the second input. This means the previous value should be shifted first and then sent to the input of the XOR gate. The encoder then stores the results in a buffer and repeats the procedure N times. After it finishes creating one N-gram, it passes the N-gram to the bundler unit, resets the buffer, and starts building the next N-gram until it hits either the last character of the input or set limit per final HD vector.

The bundler takes N-grams and adds them to a global HD vector that presents each position with a counter instead of only one bit per position. It then repeats this operation for M N-grams. Finally, the bundler applies a threshold (T) and makes a final binary HD vector representing all the processed characters while building this vector. At this point, the encoder is done. It passes the results to be stored as prototype HD vectors or used as query HD vector in AM and resets both the integer-based and binary HD vectors.

5.4.4. ASSOCIATE MEMORY (AM) DESIGN

The AM unit is implemented using PCM arrays and their corresponding circuitry, similar to the IM unit. This unit takes the output of the encoding mechanism (an HD vector) as input. Equation 5.2 shows that for the classification, we need to count the differences between the query HD vector and each prototype HD vector and then decide whether or not it can belong to the corresponding class. Although one can realize this in hardware by performing XNOR operation between the two vectors followed by a pop-count operation all in the periphery, such design requires the pop-count operation even after the XNOR, which introduces an enormous area cost and significant delay ($log_2 D + 1$ cycles). However, Demeter proposes a new column-major data mapping and intelligent data duplication for this unit and exploits the characteristics of the PCM substrate to solve all these problems for HDC-based classification. It is well-known that memristor-based memory technologies can perform vector-matrix multiplication. Therefore, Demeter

implements the required XNOR and following pop-count operations in Equation 5.2 in four steps.

Step 1: Demeter stores one prototype HD vector (or a chunk of one HD vector) in one column and its complement in the same column number of a second array. Figure 5.5-A shows their placement in the AM unit. **Step 2:** Demeter applies the query HD vector (Q) to the rows of the first array and the complement of the query HD vector ($\bar{Q}$) to the rows of the second array with complement prototype HD vectors (Ps), shown in Figure 5.5-B. **Step 3:** Demeter enables columns consecutively and effectively read out the number of ones in $Q.P$ and $\bar{Q}.\bar{P}$ in ADCs of each array. This way, it performs two vector-matrix multiplications using Kirchhoff's law, one between Q and all Ps in the first array and one between $\bar{Q}$ and all $\bar{P}s$. Section 5.4.5 describes **Step 4** that realizes XNOR and pop-count operation simultaneously. Figure 5.5-B presents a high-level illustration of AM design.

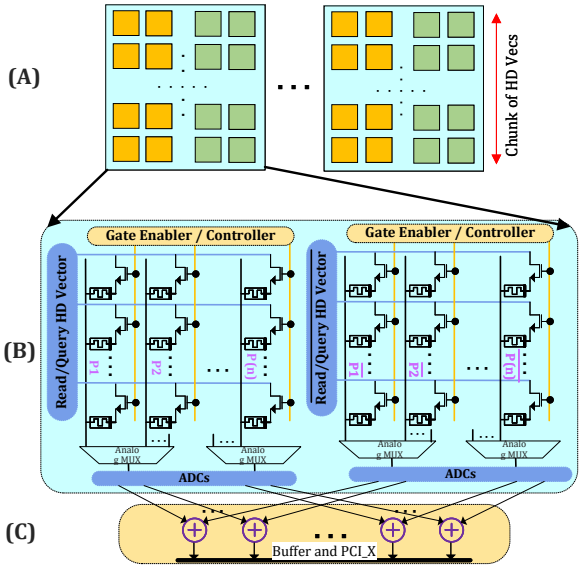


Figure 5.5: (A) Data mapping and placement of prototype HD vectors in , (B) High-level design, and (C) Partial hardware for Similarity Check unit.

Similar to the case in IM, the limited array size of PCM substrates also prevents Demeter from storing a full HD vector in one row or column of AM. To reduce the required area, and since the encoding is the bottleneck and not the classification (Section 5.5), in the AM, unlike IM, Demeter stores the chunks of HD vectors in the same array. Figure 5.5-A takes a color-coding approach and depicts the way Demeter breaks prototype HD vectors into multiple chunks and stores them in columns of AM in and among tiles. It is worth noting that Demeter only writes to the PCM cells once in both IM and AM units unless either the configuration file in Step ❶ or the user the default reference genome database in Step ❷ changes. This prevents many writes to the devices, which still have limited endurance compared to traditional

memory technologies.

5.4.5. SIMILARITY CHECK HARDWARE

The similarity check unit is a small computing unit that takes the two ADCs' output of similar columns from the two crossbars and adds them together (**Step 4**). Figure 5.5-C depicts all the logic for this unit. The output of this unit is the results of XNOR and pop-count together. At this stage, the similarity check unit sends the results out to the host CPU to determine whether the similarity is close to the threshold and should be considered in the abundance estimation (❹-b, and ❺).

5.4.6. CONTROLLER UNIT

The controller orchestrates all the operations of Demeter by generating control signals for other components. The controller is designed as a simple FSM machine and operates based on parameters set in Step ❶.

5.5. EVALUATION

5.5.1. METHODOLOGY

We emulate the execution of Demeter using a cycle-accurate RTL model and synthesized it using UMC 65 nm technology node. PCM prototypes and analytical models used for validation and further simulations are based on the results of EU project MNEMOSENE [19], led and concluded by TU Delft in 2020. Table 5.1 shows the other parameters of our PCM crossbars.

Technology	PCM (512*2048 @1bit), Cell Size = 50 F^2
Current on Conducting Devices	0.1 µA
Read Voltage	0.1 V
Read/Write Latency	Read=2.8 ns, write=100 ns
ADC	9 bits resolution, 2 ns, 4 pJ per sample

Table 5.1: PCM configuration.

Accuracy Metrics. We capture the four fundamental rates from a (food) profiler when considering the presence and absence of each species in the output, i.e., True Positive (TP), False Positive (FP), False Negative (FN), and True Negative (TN) Rate. Based on these rates, Demeter reports two standard metrics of Precision and Recall [199, 212, 213] to assess the accuracy of our (food) profilers.

Performance Metrics. Performance analysis consists of three experiments: (1) Query time, and (2) Query throughput or speed. We should emphasize that build time is normally a one-time job and does not affect the overall profiler's performance. Query time is simply the required time for profiling one single read. However, throughput is measured by million reads per minute ($\frac{MR}{m}$) and should be differentiated as it can get affected easily by other factors such as the size of the data structure, or the classifier's parallelization capability.

Datasets. We have two sets of datasets. (1) Genome sequences used as a reference database. (2) Genome sequences used as food samples and input queries. We consider AFS20 and AFS31 [195, 196] as our reference genome datasets. These datasets are two-commonly used datasets consisting of 20 and 31 food-related reference genomes related to animals whose sizes vary from 12 MB to 14 GB.

Baselines. We compare Demeter against MetaCache [195] (the most accurate food profiler) Kraken2 [199], Kraken2+Bracken [214], and CLARK [215], the top 3 alignment-free and fastest metagenomic profilers that are also commonly used for food profiling.

5.5.2. ACCURACY ANALYSIS

We have evaluated the baselines and Demeter on the species levels over AFS20 for Kylo and Kal food samples for both the precision and recall accurcy metrics. This accuracy is obtained assuming there are not any non-idealities in the hardware. We observe that Demeter stands very close to the most accurate profiler, MetaCache, and has only 1.4% and 2.6% less precision and recall, respectively, for KLyo samples. Moreover, Demeter achieves similar results on AFS31 and Kal samples. Note that accuracy is very much data-dependent, and indeed this accuracy drop is acceptable for a food profiler. We conclude that Demeter is accurate and achieves high precision and recall for food samples. These results show that Demeter's HDC-based classification approach followed by our abundance estimation technique does not hurt the accuracy of the profiler compared to baselines.

5.5.3. DEMETER'S PERFORMANCE ANALYSIS

This section compares the performance of SOTA profilers compared to our CIM-enabled accelerator design of Demeter.

QUERY TIME.

Figure 5.6 presents the time that each profiler takes to query one (short) read from the query food sample and classify its specie(s) over AFS20 and AFS31. We make two key observations. Demeter improves the query time by ~74x/88x and 272x/350x compared to Kraken2 and MetaCache, respectively, on AFS20/AFS31. This shows that the acceleration of Demeter pays off and finally makes Demeter not only an accurate but also a fast food profiler. Second, the query time for Demeter remains almost the same for both databases and does not change much. We further investigate this and realize a bottleneck shift: Step ❺ or abundance estimation that is being performed inside the CPU is now the bottleneck of Demeter. This happens because of the high-frequency Demeter achieved. However, this contrasts with other profilers that spend most of their time querying their massive data structure. For the sake of completeness and comparison, we should mention the build time on average, is around 4 minutes.

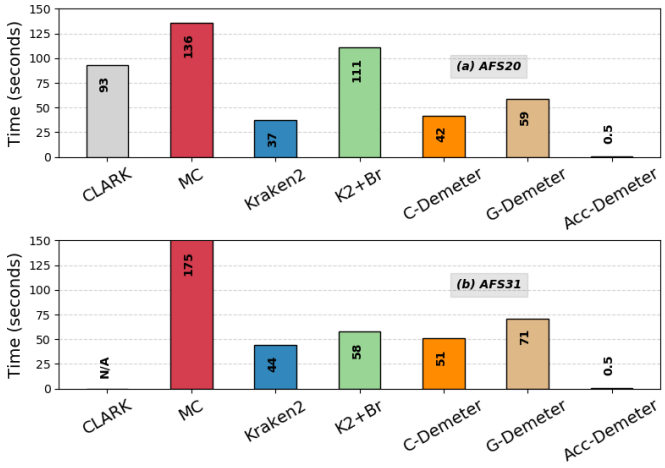


Figure 5.6: Query time on (a) AFS20 and (b) AFS31.

5

5.5.4. DEMETER'S POWER AND AREA ANALYSIS

Table 5.2 provides the area and energy consumption breakdown of different components in Demeter per query on AFS31. We make two observations. First, the logic for the encoder unit is the most energy and area-hungry unit among all others, more than 90% and 78% energy and area of the whole Demeter. This is expected because (1) the encoder consists of many CMOS circuits, whereas AM and IM are small memory units with PCM technology, and (2) the encoder is in the heart of all operations in Demeter, and we spend most of our time in this unit. We argue that this amount of logic around our array is still justifiable. Second, compared to the die area in an Intel Xeon E5-2697 CPU [216], Demeter only has an area overhead of less than 2%. We conclude that Demeter is low-cost in terms of die area.

Unit	Area (mm^2)	Area (%)	Energy (nj)	Energy (%)
IM	0.07	3.1	1.179E-06	7.4
Encoder	1.375	78.3	1.43E-05	90.6
AM	0.15	8.4	2.47E-07	1.56
Similarity	0.1815	10.2	6.91E-08	0.4

Table 5.2: Area and power breakdown of Demeter.

Our evaluations show that Demeter is capable of performing 9.45Mbp query per joule. Unfortunately, measuring the energy consumption of other profilers and having an apple-to-apple comparison between the energy consumption of this method with other ones is hard. However, Merelli et al. [217, 218] show that running Kraken2 with querying an even smaller data structure built from a reduced reference genome dataset, minikraken [217, 219], can incur more energy (maximum of $0.6\frac{Mbp}{j}$). This considerable difference happens because of three reasons: (1) Kraken2 queries a more complex data structure compared to Demeter and requires more

complex operations, (2) Kraken2 queries a bigger data structure for its query, and (3) Kraken2 incurs significant data movement between the memory and the processing unit. All of these limitations exist in similar forms in CLARK and MetaCache. We conclude that Demeter is more energy-efficient than all four SOTA baselines.

5.6. CONCLUSION

In this chapter, we focus on bioinformatics and designing a CIM-based accelerator for food profilers. Exploiting the concept of CIM, a PCM-baed hardware accelerator is designed. While Demeter maintains the accuracy of state-of-the-art profilers, it improves the energy efficiency and latency by one and two orders of magnitude. This application demonstrates a new dataflow as the communication between the memristor crossbars was minimum compared to the applications discussed in the previous chapter. The output of crossbars is mainly giving back as input to itself, considering the encoding mechanism. We learned that there is no access pattern to the rows of IM. Instead, the rows of crossbars are selected randomly. Besides, we see the necessity of new operations in the periphery, logical XOR, shift, and thresholding. These are important lessons when we want to move toward a generic approach and design a programmable CIM-based accelerator.

6

TOWARD PROGRAMMABLE DESIGN: TILE STRUCTURE AND ISA

To move toward a more flexible and generic approach to memristor-based CIM, we present a new (micro) instruction-set architecture (ISA) to control a single CIM-Tile comprising the analog memory array and all necessary analog and digital periphery. The newly introduced ISA provides flexibility in programming new CIM functionalities by simply rescheduling the instructions from the ISA. We defined our own compiler that can translate CIM-tile operations to a sequence of instructions from our ISA based on the system configuration. Besides, we discuss the detailed implementation of the CIM-Tile to see how they are controlled by the (micro) instructions. To simulate the CIM-Tile and perform design space exploration considering different technologies and parameters, we also introduce our fully parameterized first-of-its-kind CIM tile simulator.

This chapter is based on [21], [14], and [220].

In this chapter, we study the CIM-Tile, the essential components, and their orchestration. We bring programmability into perspective by introducing instructions. This is an important step toward a generic CIM platform. Figure 6.1 shows the focus of this chapter.

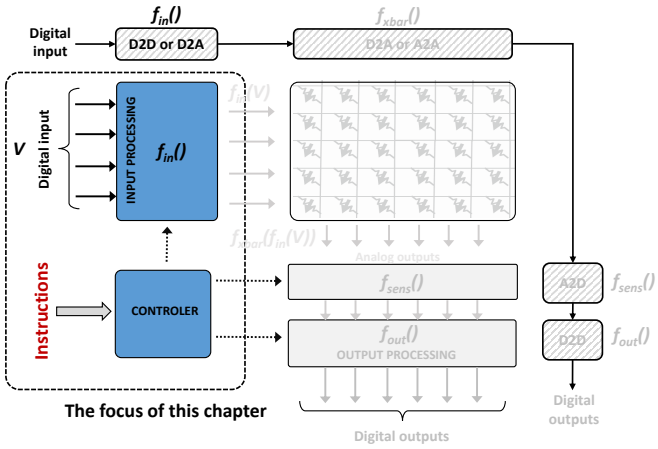


Figure 6.1: In this chapter, we mainly focus on the highlighted parts in our generic illustration of a CIM-tile.

6.1. CIM TILE INITIAL STRUCTURE

In our perspective, a CIM tile can be seen as an off-/on-chip component from the CPU. In this way, CIM tiles are not integrated into the memory hierarchy of the system and are allocated into different address spaces. This simplifies system integration (in the future) since the designer is not concerned with memory coherency. In this section, we focus on the structure of a singel CIM tile and how it is organized. First, we describe the initial version and its limitation. To address that, we explain the extended version. Finally, we elaborate on possible ways to perform pipelining inside a tile.

6.1.1. TILE OVERVIEW

Figure 6.2 depicts the architecture of the CIM-tile, including digital and analog components as well as control and data signals. The operations that can be executed on the crossbar are divided into two main categories: 1) write and 2) read/computational operations. The computational operations include *addition, multiplication*, and *logical operations*. In the following, we will describe our tile architecture and its main modules considering these two categories: (the discussion of the control signals is left to the subsequent section).

1. **Write operation**

Before doing any computation on the crossbar, the memristors in the crossbar(s)

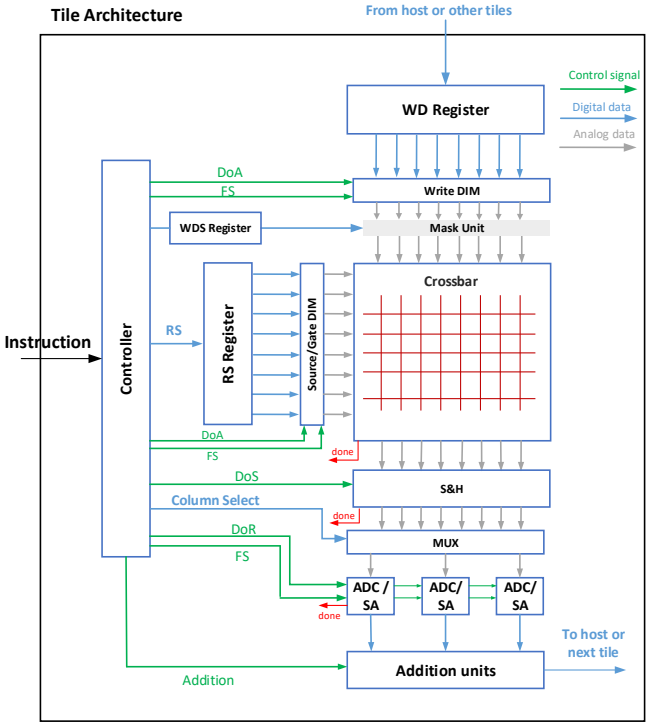


Figure 6.2: The overall tile architecture.

have to be programmed. In the case of the 1T1R crossbar structure, the memristors located in the same row can be programmed in parallel. Otherwise, sequential programming is required. Figure 2.4 provides more detailed information on programming memristors in different crossbar structures. In order to write data to the memristor crossbar, we have to specify the location in the crossbar (based on the index of row and column) where the data has to be written to. Therefore, three registers are employed to capture this information. 1) The data itself has to be written to the *Write Data (WD)* register, whose length depends on the width of the crossbar as well as the number of levels supported by the memristor cells. Considering endurance issues and potential energy savings, it is not always necessary to write data to all the array columns. 2) For this purpose, the *Write Data Select (WDS)* register is used to select which columns should be activated. This is especially relevant when considering implementing a write-verify operation. 3) Finally, the *Row Select (RS)* register is employed to activate the row in which data has to be written. In this version, the data required to fill these registers is embedded into the institutions. A more detailed description is presented in the following.

Voltages that have to be applied to the crossbar depend on the crossbar technology,

and they are usually different than the voltage used for the digital part of the system. Therefore, we need a device to convert the information from the digital to analog domain called Digital Input Modular (DIM). Selecting a row requires that two different voltage levels have to be provided for both the source and world-line of the target row as depicted in Figure 2.4(b), which means two DIMs (*Source/Gate DIM*) are required to drive both of them. Therefore, considering one DIM for the crossbar columns (*Write DIM*), we need three DIMs in this architecture in total. Based on the operation and the data stored in the RS and WD registers, DIMs can apply proper voltage levels to the crossbar. In addition, the data in the WDS register is used by the *Mask unit* to prevent extra switches for the cells that the data has not to be written into them.

2. **Read and computational operations**
In this category, the operations generate an output and it has to be read by the periphery circuits in the architecture. The generated output can be the outcome of either a normal memory read or computational operation. In contrast to the write operation, there is no need to fill the WD and WDS registers. The RS register is again used for row activation. However, among computation operations, matrix-matrix multiplication (MMM) is different than others in the sense that the RS not only has to indicate the active rows, but also can be considered as the data for one of the matrices. When the operation is performed inside the crossbar, the generated analog output has to be captured by the *Sample & Hold* (S&H) unit. This allows for a clear separation of the execution within the array and the read-out circuitry, which can be used for the pipelining of the system.

When the S&H module has captured the result from the array, the ADCs (or the sense amplifiers) can be used to convert the analog results into the digital domain. Since ADCs consume much energy and area, usually, it is not possible to allocate one ADC per column. Therefore, we need analog multiplexers to share several columns with one ADC. Besides, certain high-level operations, e.g., the integer MMM, require additional processing steps and these are performed in the *Addition Units* that are specific to the integer MMM. Our scheme for this unit is described in Chapter 7 where the design utilizes minimum size adder to impose as less latency/power as possible to the system. The design considers technology/circuit/application-driven restrictions such as the maximum number of active crossbar rows, number of ADCs, and datatype size. When other (high-level) operations are needed in the future, this unit can be altered or substituted with others.

6.1.2. CIM TILE (MICRO) INSTRUCTION SET ARCHITECTURE

As discussed before, a complex sequence of steps needs to be performed in the CIM tile that can be different depending on the (higher-level) CIM tile operation, e.g., read/write, dot-matrix multiplication, Boolean operations, and integer matrix-matrix multiplication [221]. Similar to the concept of microcode, we introduce a (micro) instruction-set architecture (ISA) for our CIM tile that would allow for different schedules for different CIM tile operations. Despite conventional instruction, these newly-defined instructions expose the hardware of a CIM-Tile to the programmer to

Table 6.1: List of initial instructions.

Instruction	Semantic	Operand
Row select	RS	data to fill RS register
Write data	WD	data to fill WD register
Write data select	WDS	data to fill WDS register
Function select	FS	data to configure the drivers/ADCs
Do array	DoA	-
Do sample	DoS	-
Columns select	CS	data for the select of MUX
Do read	DoR	-

provide high flexibility in the execution of a variety of operations. The “Controller" in Figure 6.2 is responsible for translating these instructions to the actual control signals (highlighted in green). The list of instructions is presented in Table 6.1. In the following, we discuss each instruction:

- *Row Select (RS):* The RS instruction is responsible for setting up the RS register (see Figure 6.2) that is subsequently used to correctly control the source and gate drivers to provide the right voltage levels for the crossbar. At this moment, the number of bits to set the RS register is as large as the height of the crossbar (which means the input precision is 1 bit). As the size of the crossbar increases, this is impractical for hardware implementations and will be addressed in the extended version of instructions. However, for simulation purposes, the impact is negligible and actually allows us to investigate the utilization patterns for the RS register.

- *Write Data (WD):* The WD instruction is responsible for setting up the WD register that is used to write data into the crossbar. Similar to the RS instruction and with the same reasoning, the instruction size (excluding the opcode) is as large as the size of the WD register. We envision that this instruction to be replaced when another mechanism is chosen to load data into the crossbar that is more hardware-friendly. For simulation purposes and in the initial version, it is now the only way to load data into the crossbar.

- *Write Data Select (WDS):* The WDS instruction sets up the WDS register to control which bits of the WD register need to be written. This allows for a flexible manner to write data into the crossbar in light of potential endurance issues associated with current-day memristor technologies. It is especially useful when a write-verify operation needs to be performed where the written data into the crossbar is compared with the golden data. With the help of this register, correctly written bits can be masked out, which helps to improve the endurance of the crossbar.

- *Function Select (FS):* The FS instruction is needed for several reasons. Functionally, the DIMs need to be set up differently when writing data into the crossbar, or when reading out data, or performing compute operations in the crossbar.

Furthermore, it is envisioned that the future periphery needs additional control signals. These are now conceptually captured in the FS instruction. For example, the control signals needed to control the “Addition units" are more than one, i.e., more than one instruction is needed to control these units. The instructions related to this unit are provided in the extended version.

- *Do Array (DoA):* The DoA instruction is used to actually initiate the DIMs after they have been set up using the RS, WD, and FS instructions. This instruction will have a variable latency depending on the operation that is performed in the crossbar. These delays are specified as parameters in our simulator. In the hardware implementation, this latency (of the crossbar) can be captured by a (programmable) counter or by a ‘done’ signal issued from the crossbar itself. More importantly, this instruction allows for a clear (conceptual) separation between the different pipeline stages within the CIM tile. This will be discussed later.

- *Do Sample (DoS):* The DoS instruction is used to signal the S&H module to start copying the result from the crossbar into its own internal storage. It must be noted that this module still operates in the analog domain. The introduction of the DoS instruction allows for a conceptual separation of the execute stage (crossbar) and read stage (ADC). After the values are copied into the S&H module, the crossbar can basically be activated by the next DoA instruction.

- *Column Select (CS):* The CS instruction is used to set up the CS register that controls the multiplexers (in the MUX module) in the read stage. In case each column of the crossbar can be associated with its own ADC/SA, there is no need to have the CS instruction. In all other cases, the CS register flexibly controls which column (in the S&H module) is connected to an ADC/SA. The length of the CS instruction is related to the number of columns of the crossbar. For the same reasons as with the RS and WDS (and WD) instructions, it is purposely defined as it is now in this initial version and implemented as such in the simulator to allow for further investigation in the future. It is expected to be optimized or replaced when a hardware implementation is considered.

- *Do Read (DoR):* The DoR instruction initializes the modules “ADC/SA" to start converting the output of the S&H module (via the MUX) into a digital representation. It is expected that multiple iterations of the CS and DoR instructions need to be issued in order to completely read out all the columns of the crossbar. It should be noted that when none of the columns allocated to an ADC/SA are selected, the sensing unit in is not activated. Depending on the complexity of the module, the latency can vary, and thus, a ‘done’ signal is needed (see Figure 6.2) to signal the end of the readout before the next DoR instruction can be issued. This instruction might also be quite practical when we have ADCs with configurable resolution. The resolution of ADC is directly related to how many times it gets activated for an input signal.

It is important to note that the crossbar, the S&H modules, and the ADC modules (related to the DoA, DoS, DoR instructions, respectively) need to be able to signal

to the controller that their operation is finished. It is only after this signal that subsequent instructions can be issued by the controller. If this turns out to be impossible in an actual hardware implementation, we envision the need to set up counters that are initialized to the specific operations to achieve the same functionality. Both approaches are already supported in our simulator.

6.1.3. CHALLENGES OF THE INITIAL DESIGN

Although the ISA defined before brings programmability and flexibility into the design, they impose significant challenges when it comes to their hardware implementation. In the following, we elaborate on the main limitations of the proposed ISA. Based on that, we present the extension of the CIM-tile structure and its ISA.

- The initial set of instructions embeds all of the data that has to be fed to the crossbar into the instruction. For example, in the case of MMM, while one operand is programmed into the crossbar, the second operand is given to the RS register to be fed to the crossbar as input. This data is embedded into the 'RS' instruction. Similarly, the data that has to be programmed into the crossbar is embedded into the 'WD' instruction. However, for real applications, the data may not be known at compile time and produced during execution. We can consider a deep neural network as an example here, where the input to the intermediate layers is generated during the execution time. Hence, assuming the data is always present in the instruction is unrealistic. Therefore, when designing the instructions set, it should be taken into account that compile-time unknown data is fed to the CIM-tile at run-time.

- The CIM-tile would be a part of a micro-architecture. From the system perspective, the data for a CIM-tile is provided either from another Tile or an external component (e.g., CPU). This data is transferred via a databus. In a realistic assumption, the bandwidth of this databus limits the amount of data that can be read per instruction. Since we have not considered any hard constraints on the size of databus, the current design is not conditioned by the fact that the databus might be too small to provide all required data in a single cycle. Hence, a proper mechanism should be defined when interfacing the tile with the outside.

- The instructions generated for an application/kernel should be stored in memory to be executed during run-time. Since the storage of the instructions is assumed to happen at compile time, the instruction memory sets an upper limit for the size of a program. Therefore, it is essential to have a reasonable size for each (micro) instruction to use available instruction memory size more efficiently. Considering the initial version of the instructions, the implementation for some of them, such as row selection, write data, write data selection, and column selection instructions, would be data-intensive. Even a small-sized MMM operation is translated to an instruction size between 0.49 and 3.4 MB, depending on the ADC precision and the number of ADCs. Hence, the instruction should be revisited

with this aspect in mind.

Table 6.2: Instruction file exploration for the Gemm benchmark

#ADC	ADC resolution	Instruction size (MB)	%RDS	%WD/WDS	%CS
8	5	13.75	3.08	0.06	96.8
8	8	1.74	3.44	0.45	95.67
32	5	3.8	11.25	0.21	88.34
32	8	0.5	12.17	1.59	84.66

To get an indication regarding the total instruction file size as well as the contribution of each instruction, we have compiled the 'Gemm' benchmark, consisting of a single matrix-matrix multiplication, preceded by storing the multiplicand matrix in the crossbar. Table 6.2 shows the contribution of the individual data-intensive instructions to the instruction file size. The exploration is done assuming the crossbar size of 256×256. As we discussed, 'RDS', 'WD', 'WDS', and 'CS' are the main contributors to the instruction file size. While all of the mentioned instructions implementations should be modified in order to attain hardware feasibility, the main data reduction can thus be expected in the change of the CS instruction implementation. It should be noted that the sequence of instruction depends on many parameters, including the number of ADC and their resolution. We explain this later. This information provides insight to steer toward the extended CIM-tile structure and (micro) ISA.

6.2. CIM TILE EXTENDED STRUCTURE

6.2.1. TILE OVERVIEW

The overall CIM-tile architecture presented throughout this section is depicted in Figure 6.3. The CIM-tile is expected to interact with external devices and therefore, a clear interface needs to be defined. The interface comprises two input buffers (Write-Data (WD) and Row-Data (RD)) and an output buffer. This allows independent CIM-tile operation without needing an external controller knowledgeable of the CIM-tile internals. Specifically, the external components can be agnostic about the CIM-Tile instructions and their status. These buffers are explained in the following:

- *WD buffer:*

  This buffer serves as intermediate storage to alleviate the timing constraints of sending data to the CIM-tile when the (bus) bandwidth is insufficient to transfer all required data in one shot - i.e., to the WD register. Therefore, each WD buffer row corresponds to a chunk of data received from the outside controller. This data is going to be programmed into the crossbar, ultimately. The WD buffer has been sized such that an entire (array) row of data can be stored inside it. Consequently, after transfer to the WD register during a write operation, the WD buffer is ready to receive the next set of data. There would be instructions to stream the data out of this buffer. We will discuss this in the following subsection. Using this buffer,

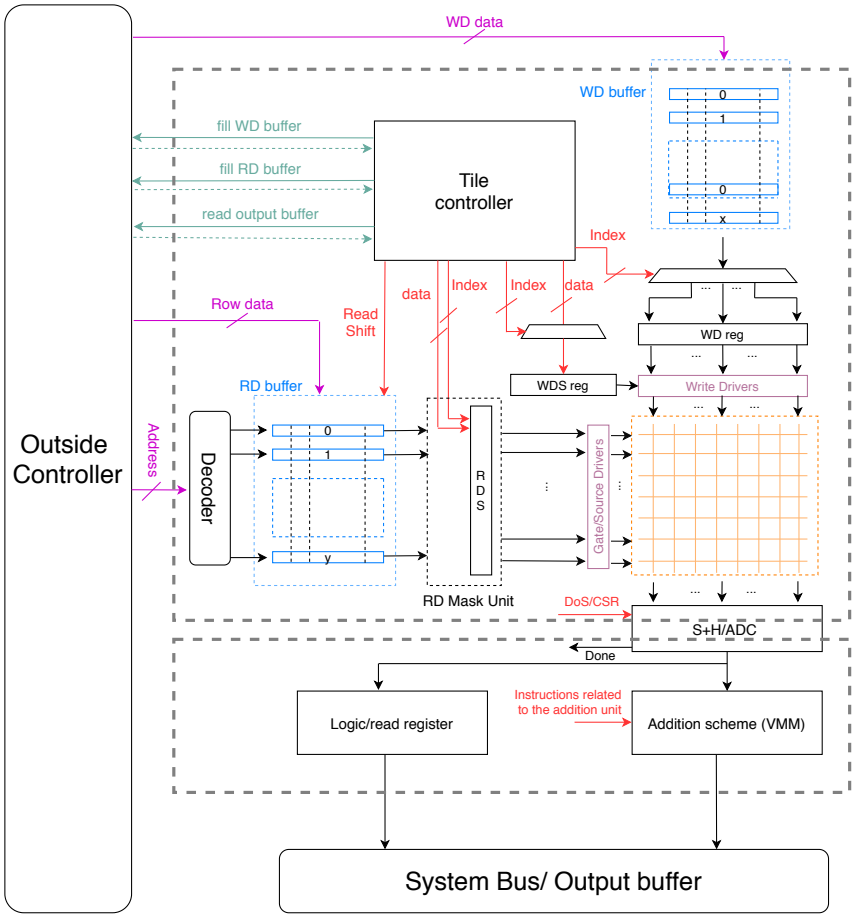


Figure 6.3: Overview of CIM-tile extended structure.

a mismatch between crossbar size and bus width can be handled. The two main architectural parameters that impact this buffer are 1) the bus width provides input to the buffer from outside and 2) the number of crossbar columns. Based on that, there would be two scenarios:

In the best scenario, the bandwidth is large enough to provide all the required data to be written into one row of the crossbar in a single cycle. This means the data is written into this buffer in a single cycle, and then the tile controller copies the data to the WD register for the next write operation. However, in a more realistic scenario, the bandwidth is small to be able to transfer all the data in one cycle. Therefore, a new design parameter is introduced for the WD buffer size. The buffer size can be either 1) equal to the available bandwidth (minimum size

buffer) or 2) equal to the crossbar width to save all the information for a single row. In the following, we discuss these two cases.

The main advantage of having a minimum size buffer is the lower area cost. However, this leads to timing constraints for filling the buffer. To clarify, consider an example where a crossbar width is 256 columns (each column one bit), and the bandwidth is 128 bits. During the first cycle, the outside unit provides the first 128-bit to the WD buffer. During the second cycle, this data will be put in the correct position of the WD register by the tile controller. In the third cycle, the outside unit will write the second 128-bit to the WD buffer. During the fourth cycle, the second 128-bit can be put in the WD register, meaning the register is full, and the tile can thus perform its write operation. Now, we consider the case of a WD buffer of size equal to the width of the crossbar (so 256-bit). We still need to spend two cycles for outside to transfer the data to the RD buffer. However, this can be two subsequent cycles as the CIM-tile does not need to process the first half. Hence, the larger the WD buffer is, the fewer timing constraints are between the outside unit and the tile controller.

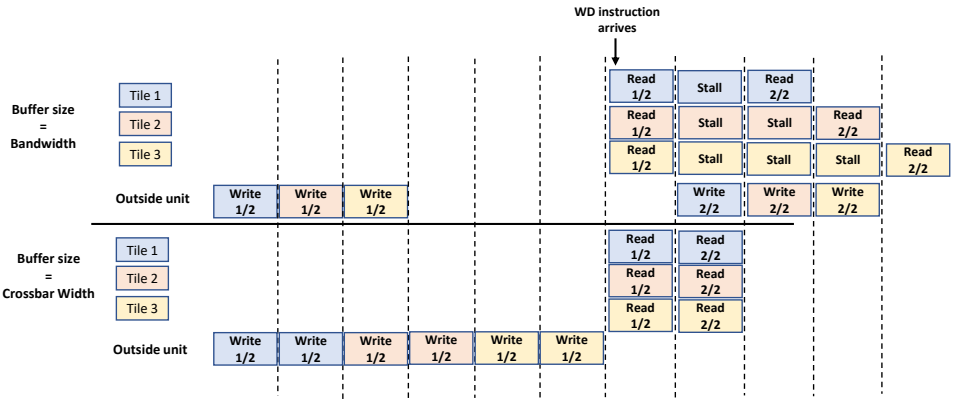


Figure 6.4: Execution flow for three tiles considering two WD buffer sizes. In this example, all tiles arrive at the WD instruction at cycle 6.

Figure 6.4 illustrates the aforementioned example where there are three different tiles. In this example, the WD instruction arrives at the same time for all three tiles. Despite the possible performance overhead of the first case (buffer size equal to the bandwidth), the major challenge is the synchronizations between the outside and the CIM-tile. The actual performance impact is dependent on a number of factors, such as the number of tiles serviced by the outside unit, the application (Number of write operations and their position in the program) and the available bandwidth for writing to the WD buffer, and the cycle latencies of other pipeline stages of the tile. Hence, in our design, we considered the WD buffer to be equal to the crossbar width. Now that the buffer size has been determined, two implementations of the buffer can be considered.

1) In the first implementation, a buffer based on the FIFO shift register may be employed, of which the width is equal to the bandwidth for writing to the buffer and the height is such that the product of width and height equal the size of the crossbar width. This solution has two drawbacks. First, as only the topmost row of the buffer can be accessed by the outside unit, a fixed number of shifts is required for the first set of data to arrive at the bottom of the buffer. Secondly, shifting the data through the entire register leads to extra switches in the buffer, leading to energy usage overhead.

2) In the second implementation, a counter that increments on a write from the outside unit, decrement on a read from the tile, and remains constant when both happen during the same cycle is placed. It can be ensured the data from the outside is written to the bottom-most free row of the buffer. The data present in the filled rows of the buffer is shifted downwards one row when the tile reads a row from the buffer. An obvious drawback of this third solution is the extra hardware that is required to allow for the individually addressable rows of the buffer and the simultaneous reading and writing. Figure 6.5 shows the design of this buffer.

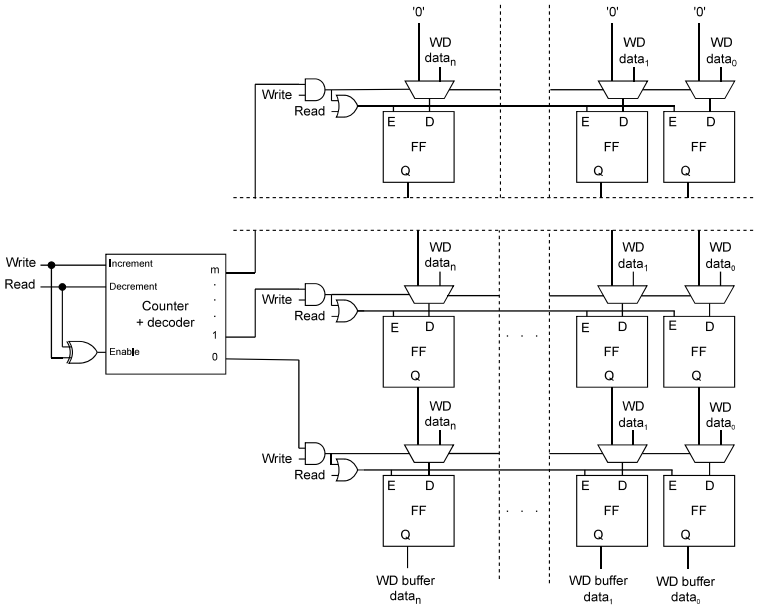


Figure 6.5: Implementation of the WD buffer.

- *RD buffer:*

  In the case of computation operations, the RD buffer holds one operand of operations and feeds it to the crossbar. Since the crossbar drivers (DAC) have limited resolutions (usually one bit), the data must be split and given to the driver

in several cycles. As this buffer feeds data into the crossbar bit-by-bit, starting from LSB to MSB during a Matrix-Matrix Multiplication (MMM) operation, this buffer is implemented using a parallel-in-serial-out (PISO) register per crossbar row. By having the width of the buffer sized to fit the maximum supported datatype size (e.g., 32-bit), the buffer only has to be filled once for each of the element dot-product operations. The height of the buffer is equal to the number of crossbar rows. After shifting the MSB out of the buffer, the next set of data can be loaded into the buffer to reduce the overhead of data transfer. It should be noted that after each shift to the right, the output of the buffer is written back to the input. This is needed when computing on a datatype size lower than the one the system is designed for. This relates to the 'Addition Scheme' and how this periphery unit works, which is discussed in Chapter 7.

In some scenarios, the data for the RD buffer is loaded from external memory (e.g., DRAM or Flash). This data is a vector of elements, and in each cycle, one (or a few) bit(s) should be given to the crossbar (due to DAC resolution). In case there is no RD buffer, when this (vector) data is loaded from external memory, it needs to be processed to extract and combined the bits required from each element and send them to the crossbar. Hence, it needs temporary storage as well as extra processing. This indicates another advantage of this RD buffer which makes the outside completely unaware of the tile limitations (e.g., DAC resolution) and removes the extra processing and the temporary storage.

- *Output buffer:*

  This buffer is employed only for temporarily storing the result data until the data has been transferred to other tiles or processors. As the output of the crossbar should be stored in this buffer, its size determines the amount of parallel computation that can be performed. Without this buffer, the outside world should be synchronized with the execution of instructions in the tile, which can increase the complexity on both sides. The output buffer also has to be sized properly. The size of the output buffer depends on the datatype size. Table 6.3 shows the size of each element generated by a crossbar and the required output buffer size based on different datatype sizes. This number is brought for MMM operation and the crossbar and assuming both operands have the same datatype size. While the result of a single element for a smaller datatype is smaller than for the maximum datatype size, more elements can be stored in the crossbar, and thus more computation can be performed. Therefore, based on the datatype sizes intended to support, a designer has to place a proper output buffer. In addition, to flexibility support different datatype sizes, more consideration has to be taken into account. Chapter 7 focus on this part.

In addition to the buffers, other digital components (depicted in Figure 6.3) are:
Tile registers: Same as the initial version, (i) the *Write Data (WD)* register contains the data that has to be written into the crossbar. The data loads from the *WD* buffer in several steps depending on the WD buffer row size. The register length depends on the width of the crossbar as well as the number of levels supported by

Table 6.3: Required output buffer sizes assuming 256×256 crossbar and 1 bit per each memristor.

Datatype size	Element size (bit)	Output buffer (bit)
32	72	576
16	40	640
8	24	768
4	16	1024
2	12	1536
1	8	2048

the memristor cells. However, due to the flexibility attained by our instructions, we can opt to only partially fill the register if the kernel does not intend to write into all of the columns. (ii) The *Write Data Select (WDS)* register indicates to the write drivers which columns should be written to. The data for this register is embedded in its instruction to provide more flexibility for kernels. Finally, (iii) the *Row Data Select (RDS)* register is employed for the activation of crossbar rows and used for all operations, including write, read, logic, and VMM. In addition, in the case of VMM, if the ADC resolution does not support the activation of all the rows, this register is used to activate batches of rows in several steps. Similar to the *WDS* register, the data for this register is embedded into its instruction. To clarify more, the data in the RDS register shows which region of the crossbar should be active, while the RD buffer stores the data for one operand of compute operations.

The other two main components are the tile controller and the 'Addition Scheme'. The controller will be discussed in this chapter, but the discussion regarding the addition unit is left for Chapter 7.

6.2.2. CIM TILE (MICRO) INSTRUCTION SET ARCHITECTURE

To execute a kernel on the tile, a complex sequence of steps has to be carried out considering different operations, the patterns of column and row selections, datatype sizes, and read-out circuitry specifications. An extension of in-memory (micro) instructions is defined with the objective of keeping the hardware simple and generic by moving the complexity to the compiler, reducing the instruction file size, and maintaining high flexibility. The list of instructions is presented in Table 6.4. The instructions are explained shortly in the following.

- Row Data (***RDxx***): The first group of instructions is related to the crossbar rows.

  Row Data Selection: In the initial version of this instruction, we dedicated one bit for each row of the crossbar to indicate whether that row should be activated or not. This causes an issue in terms of the instruction size. To address this, two solutions can be provided. In the first solution, which is called row-wise row selection, we only pass the index of a few rows into the instruction. In case the instruction allows us to embed 24 bits of data into it (excluding opcode) and the crossbar size of 256, we can only address three rows at a time ($3 = 24/log2(256)$). Hence, multiple of this instruction should be used to activate more rows. The

Table 6.4: Overview of the new ISA

Opcode	Op 1	Op 2	Function description
RDSb	Index	Mask	Place 'Mask' into RDS reg at 'Index'
RDSc			Clear the entire RDS register
RDSs			Set the entire RDS register
RDsh			Shift RD buffer contents
WDb	Index		Copy data to for WD buffer to WD reg. at 'Index'
WDSb	Index	Mask	Place 'Mask' into WDS reg at 'Index'
WDSc			Clear the entire WDS register
WDSs			Set the entire WDS register
FS	Function		Select crossbar functionality
DoA			Activate the crossbar function
DoS			Sample the crossbar output
CS	Index	Activation	Select column to be read by ADC
DoR			Activate the ADC
jal	Address		Jump to 'Address' and store PC.
jr			Jump to the PC stored in return reg.
BNE			Branch to PC stored in branch reg.
LS			Indicate the last section of rows to reads
IADD			Activate third stage of the addition unit
CP			Copy result per ADC to output buffer
AS	Selection		Select adders for addition between ADCs
CB			Copy the result of addition between ADC to output buffer

advantage of this is activating any random rows in case the application does not follow any pattern. However, this solution would be inefficient for selecting a large number of rows as the instruction count will scale linearly with the number of rows to be selected. In addition, the implementation of this row-wise solution would lead to multiple large decoders.

The second solution of row selection aims to group the rows and indicate which rows are activated within a specific row. This solution is called block-wise row selection. As an example, the crossbar with 256 rows can be grouped into 16 parts; each contains 16 rows. The instruction first identifies the group index ($4bits = log2(16)$) and then provides 16 bits of data for each group. We refer to this instruction as **RDSb**. This solution is more efficient if consecutive rows have to be selected. For example, if only rows from index 0 to 16 should be selected, this can be done by clearing the RDS register and executing a single RDSb instruction. To allow for clearing the register, a new instruction is introduced; RDS-clear (**RDSc**). In case most rows have to be activated, we can set the entire RDS register and use RDSb instruction to deselect the rows that should not be activated. To enable setting the entire RDS register, another instruction is introduced; RDS-set (**RDSs**). Being able to select and deselect the rows in the crossbar is quite essential, especially when the precision of ADC is not enough to activate the entire crossbar rows. In this case, we use this instruction to split the rows into groups and activate them sequentially. It should be noted that ADC precision is an important parameter, and it even has a consequence on the program's size.

Finally, **RDsh** will shift the *RD* buffer to the right to present the next bit for a new compute operation. This instruction is quite useful, especially when we are dealing with datatype sizes smaller than what the system is designed for. When

the datatype size in the crossbar is smaller than what a system is designed for, the same input data has to be given to the crossbar for several iterations due to the resource limitation in the periphery (Chapter 7). This is why, after each shift, the output of the RD buffer is written back to its input. Figure 6.6 depicts the state of the RD buffer after each shift. Since the data size (highlighted in blue) is half the system datatype size, the same data has to be given two times to the crossbar. However, at time step T4, we need to shift the data 4 times to the right to position it correctly. This is done by executing 4 RDsh instructions in a sequence. Employing this instruction, we can move around the data in the RD buffer with high flexibility. Figure 6.7 illustrates the overview of the underlying hardware for all the instructions related to the row data.

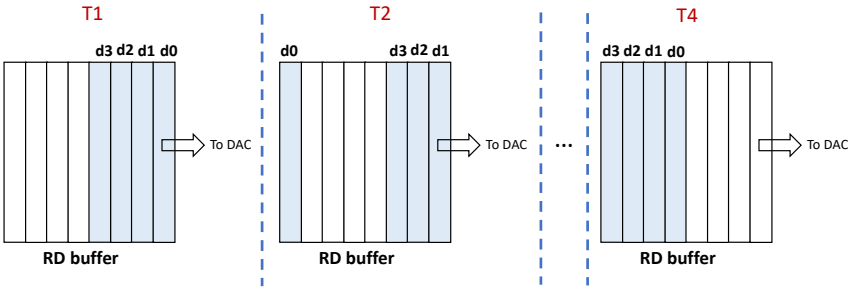


Figure 6.6: The state of RD buffer when the datatype is smaller than the maximum supported datatype size of the system.

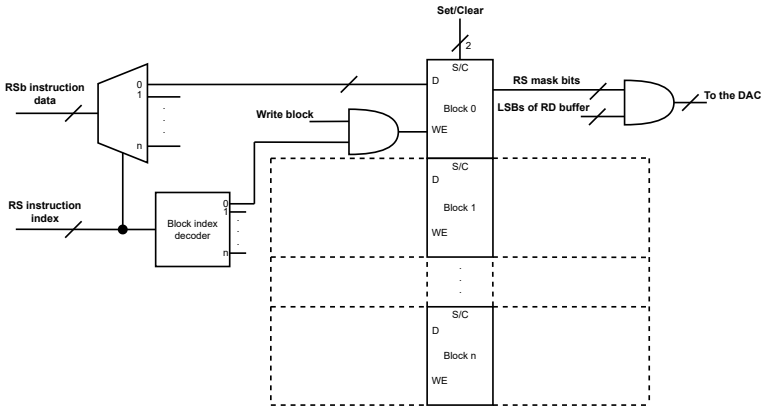


Figure 6.7: Instruction relates to row data and their underlying hardware.

- Write Data (**WDxx**): The **WDb** instruction indicates that the data present in the last register of *WD* buffer has to be moved to a section of *WD* register determined in the instruction by 'index'. If we only want to program part of a crossbar row,

this instruction helps us to navigate this information from the WD buffer to the correct location in the WD register. Like row selection, **WDSb** instructions load the data to the *WDS* register. Similar reasoning as for the RDSb solution applies to the write data selection, leading to a block-wise solution for the WDS instructions as well, introducing two new instructions to clear and set the entire WDS register; Write-Data-Select-clear (**WDSc**) and Write-Data-Select-set (**WDSs**), respectively.

- Crossbar (***FS, DoA***): To operate on the crossbar, first, we need to configure the drivers to provide proper voltage levels based on the desired operation. This is done using the Function Select (*FS*) instruction. This instruction is also used to configure the read-out circuitry as well as bypassing the *RD* buffer data in the case of read/write/logical operations. By decoding the *DoA* instruction, drivers are activated, and the operation is started on the crossbar.

- Read Out (***DoS, CS, DoR***): Due to the overhead of Analog-to-Digital Converters (ADCs), they need to be shared between several columns, which translates to the necessity for a *Sample-and-Hold* unit to save the crossbar's output. The *DoS* instruction activates this unit at the right time. The data can be sampled when the second stage is already done with the prior sampled data.

  The initial implementation of **CS** offers large flexibility at the cost of huge instruction size. Figure 6.8 depicts how this initial version works. In this example, the compiler generates the code for a small crossbar size of 32 × 32 with 4 ADCs. This means each ADC shares 8 columns. In each cycle, one column from each group is read by an ADC. Due to the instruction size and its frequent recurrence, column select instruction makes up for about 88% of the data in the benchmark. To address this, in the new version of CS instruction, we pass a single index in the CS instruction, which will set all ADCs to that specific index and use a set of activation bits to indicate whether an ADC should read the column at that index. Considering the above example, the size of CS instruction (excluding opcode) is reduced from 32 bits down to 7 bits ($log2(8)+4$). Finally, similar to the initial version **DoR** instruction activates the conversion.

- Jump instructions (***jal, jr, BNE***): As the read stage often performs identical sets of instructions (e.g., when performing a MMM, the same set of columns has to be read for every dot-product operation), a jump instruction is introduced to save a large portion of the instruction file size. Similarly to MIPS, jump-and-link (**jal**) and jump-register (**jr**) instructions are introduced, allowing for re-using the same block of instructions for every read. The 'jal' instruction stores the current program counter and jumps to the set of instructions (often readout). In order to correctly resume the instructions after branching, a 'jr' instruction should be introduced, which allows for returning to the program counter stored by the 'jal' instruction. Branch Not Equal (**BNE**) instruction is used for write verification. Due to the memristor non-idealities, sometimes we need to verify what we programmed into it to ensure its correctness. This instruction performs this task for us and jumps back to repeat the programming if the operation is not verified.

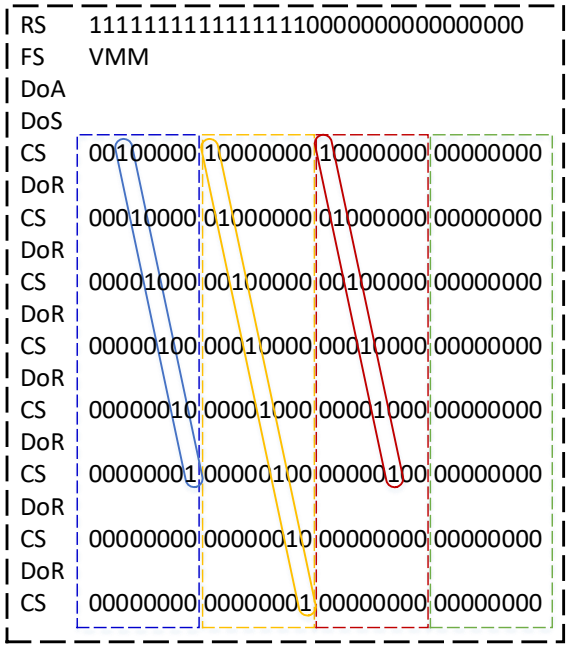


Figure 6.8: An example of instructions generated by the compiler using the initial version of the CS instruction.

- Addition unit (***LS, IADD, CP, AS, CB***): According to the structure of the addition unit, which will be discussed in Chapter 7, in the case of low ADC resolution, the *LS* instruction indicates the last section of rows (multiplier) that are activated. The *IADD* instruction performs an addition between different bits of the multiplier. If a number is distributed over more than one ADC, *AS* instructions carry out the addition between the results taken per ADC. Finally, *CP* and *CB* load the result to the output buffer obtained either per ADC or between ADCs, respectively.

Compared to the initial instructions, we could also reduce the instruction size, which is achieved by a more efficient definition of our instructions. Table 6.5 compares the total instruction size for our initial definition and the new version. This improvement is achieved mainly due to two reasons: 1) A more compact definition of column select instruction and 2) Introducing the Jump instruction to reduce the repetition in the code. It should be noted that the numbers in the table are for Gemm benchmark consisting of many VMM operations. The current version of the compiler cannot detect whether the sequence of VMM operations follows the same pattern. However, if it can, only the instruction needed for one VMM with an approximate size of a few hundred Bytes can be stored and reused for other VMM operations.

Table 6.5: Instruction file exploration for the Gemm benchmark

#ADC	ADC resolution	Initial instruction size (MB)	Final instruction size (MB)
8	5	13.75	0.28
8	8	1.74	0.11
32	5	3.8	0.28
32	8	0.5	0.11

6.3. PIPELINING

In order to improve the performance, the CIM-tile can break into a few stages which can be active in parallel. This means that the instructions associated with them can be executed in parallel, which leads to higher performance. In the following, we elaborate more on four possible stages in the tile (indicated by different colors in Figure 6.9).

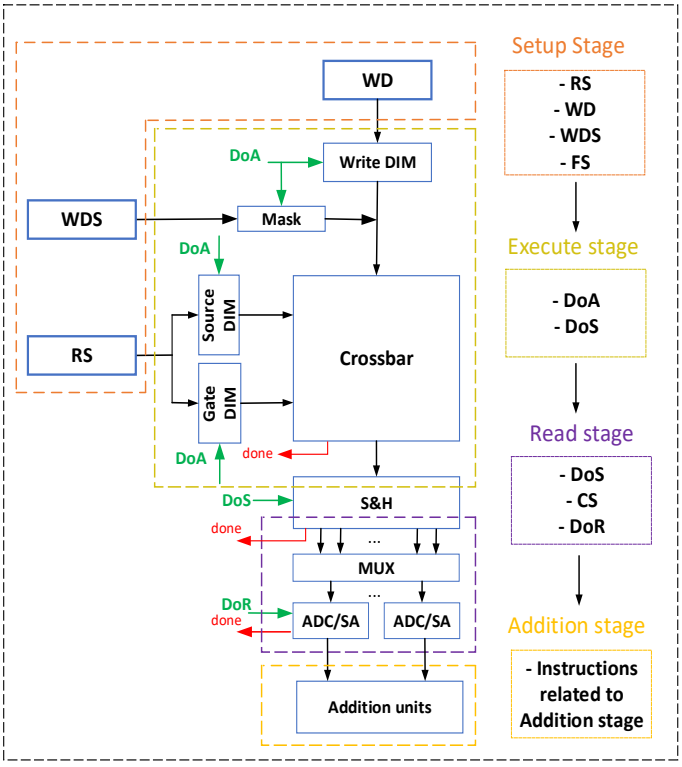


Figure 6.9: Pipelining of the crossbar CIM-tile.

1. Set up stage (digital): all the control registers (and write data register) are initialized

2. Execution stage (analog): perform the actual operation in the analog array

3. Read out stage (analog): convert the analog results into digital values

4. Addition stage (digital): perform the necessary operations for the integer matrix-matrix multiplication

These stages sequentially follow each other while performing higher-level operations translated to a sequence of instructions in our ISA. It should be clear that the pipelining described here is different from the traditional instruction pipelining. In the latter, the latency of each stage should be matched with each other in order to have a balanced pipeline. In the CIM tile, the latency of the operation performed in the analog array is expected to be much longer than the latency of a single clock cycle in the digital periphery. Therefore, it is important that the right signaling is performed between the stages in order to enable pipelining. The introduced execution model to pipeline the operations within the CIM tile will allow for trade-off investigations between different NVM technologies and the (speed of the) digital periphery. Considering the aforementioned stages, the designer should evaluate the latency of each stage (which depends on the configuration of the tile, memristor technology, etc.) to realize the contribution of each stage to the total latency of the tile and merge some of them in the case there is a stage which has by far less latency than others. This analog/digital behavior of the CIM-tile restricts the choices and their effectiveness regarding the pipelining stages. It is worth mentioning that the two analog stages (*Execution* and *Read out* stages) cannot be split into more stages. This is the same for the *Set up Stage* as well since the registers initialized here cannot be changed before activating the crossbar to ensure the correct functionality of the system.

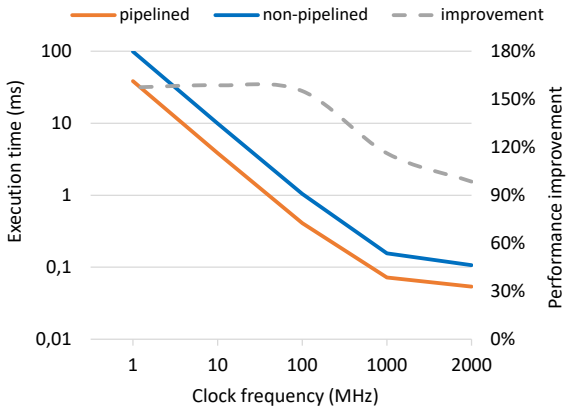


Figure 6.10: Performance improvement due to the unbalanced pipelining of tile used ReRAM/PCM device for GEMM benchmark

In the following, we provide an initial investigation of the CIM-Tile pipelining. First, we evaluate the effect of clock frequency in a CIM-Tile comprised of analog-digital circuits. The latency of the operations in the (analog) crossbar array is a constant

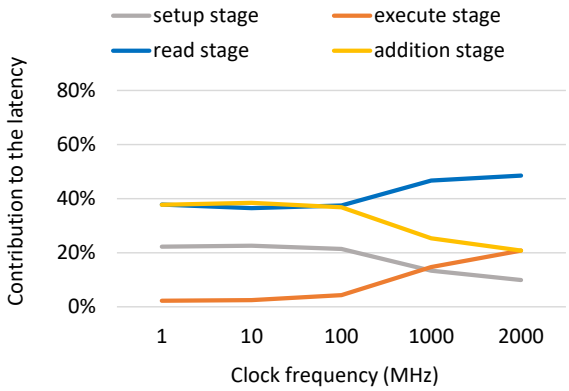


Figure 6.11: Contribution of each pipeline stage on the latency of the tile considering different clock frequencies

number. Therefore, it is interesting to determine how fast the digital periphery should be clocked in order to 'match' this latency in order to make the pipeline more balanced. In the following investigation, we have fixed the number of ADCs to 16 and ran the GEMM benchmark at different frequencies. Figure 6.10 clearly shows that performance improvements can be gained by raising the frequency of the digital periphery. However, increasing the clock frequency beyond 1 GHz does not result in much better execution times as the analog circuits (relatively) are becoming the bottleneck. In addition, the performance improvement due to the pipelining will be reduced since the stages are more unbalanced. A positive side-effect is that pipelining more balanced stages will usually lead to better performance improvements over a non-pipelined design.

Second, Figure 6.11 depicts the latency breakdown consumed in each of the 4 stages against the clock frequency. Since several columns share an ADC, reading all the columns should be performed in multi-steps. In addition, after each ADC activation, digital processing is required in *'addition unit'*. Therefore, we can clearly observe that with a low frequency, the read and addition stages are completely dominant in the total latency. Using an efficient structure and minimum-sized adder, the latency of the addition unit is no longer than the read stage. With low clock frequency, the latency of the analog circuits is hidden in one clock period. However, As the clock frequency increases, the latency of the analog components starts to rise (relatively). Consequently, we can observe that the latency of the read stage, which is composed of analog (latency of ADC) and digital (decoding latency) latency, as well as the execute stage impose more latency. This information can be used to determine the number of pipeline stages for the actual implementation.

Third, Figure 6.12 depicts the latency breakdown in each of the 4 stages against the number of utilized ADCs. It should be clear that with increasing the number of ADCs, the number of cycles spent in the readout stage is greatly reduced. Consequently, we can observe that the relative contribution of the setup stage to the

total latency grows accordingly. The contribution of the other stages to the total latency is almost negligible. With the advent of advanced ADC design to have more ADCs per tile, this figure brings insight into its implications and helps future design decisions.

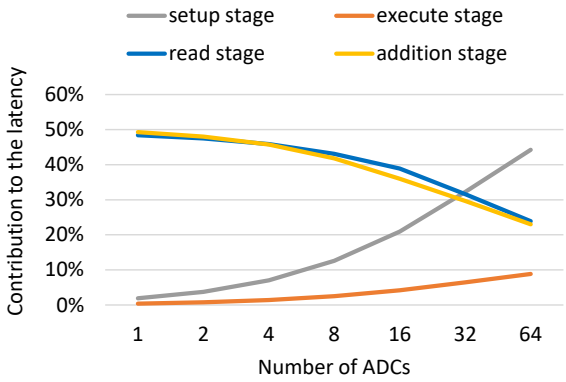


Figure 6.12: Effect of number of ADCs on the latency of pipeline stages in 100 MHz clock frequency

6.4. CONTROLLER

This section describes the design of the tile controller. The controller serves two main purposes, namely 1) executing the nano-instructions from the compiled program, and 2) communicating with the outside unit when the WD/RD buffers need to be filled or the output buffer should be read. Figure 6.13 depicts the overview of the tile controller assuming two pipeline stages (by merging the first and second stages into one as well as the third and fourth stages into another).

6.4.1. INSTRUCTION MEMORY AND DECODER

As discussed before, despite traditional pipelining, here, each stage is associated with some instructions, and they are executed in parallel. Considering two pipeline stages, two instruction decoders are required. As the different stages correspond to separate, non-overlapping subsets of the ISA, the two decoders do not have to support all different instructions but just the instructions that correspond to their respective pipeline stage. This simplifies the hardware complexity of the decoder. Figure 6.14 shows how the instructions are stored in the instruction memory.

6.4.2. STALL DETECTION

The CIM-Tile may consist of two or more pipeline stages. These stages depend on each other and have unbalanced latency. Hence, there should be a unit to synchronize these stages. The stall detection uses the information provided by the

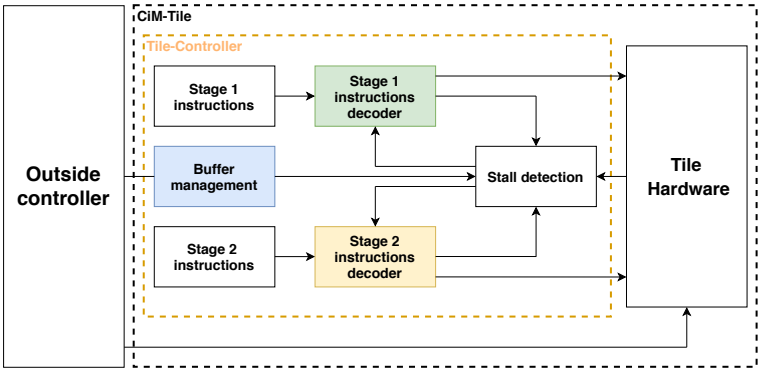


Figure 6.13: High-level CIM-tile controller to support two stages.

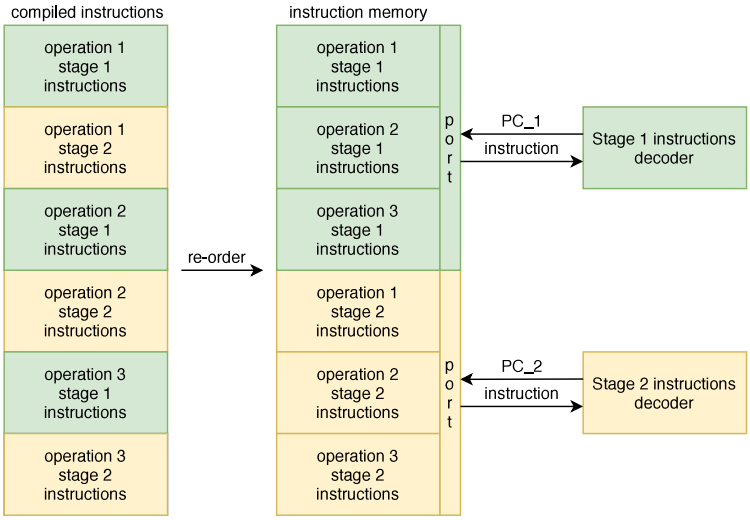


Figure 6.14: The overview of instruction memory.

opcode of the instruction at the current PC of each stage to synchronize the stages to allow for proper program execution. Considering the two pipeline stage example, the first stage should be stalled in a few cases, such as 1) when valid data is not available in RD/WD buffer or 2) when the second stage is not ready to receive new data (still busy with reading from S&H), or 3) when stage 2 is performing a read for write verification. Similarly, the second stage should be stalled when 1) the first stage has not yet finished its operation instructions (indicated by DoS) or 2) a value should be written to the output buffer while the outside unit should still read the current values present in that buffer. The stall detection unit should be designed carefully to consider all possible scenarios.

6.4.3. BUFFER MANAGEMENT

The buffer management block sends signals to the stall detection and the outside unit providing information on whether there is valid data present in each of the buffers. The way the state of the buffers is monitored differs for each of the three buffers.

- Output buffer: The content of this buffer is valid after the execution of instructions that activate this buffer. This is either the CP or CB instruction for an architecture without or with the addition between the ADCs stage of the addition unit, respectively. For the read and logic operations, the output buffer is activated after the final read has been performed. The buffer management receives a signal from outside when the content of the buffer is read. Then it notifies the stall detection unit to allow for the next computation result can be stored in the buffer. This unit also sends a signal to the outside whenever the buffer is written with new information.

- WD buffer: We track the state of the WD buffer using a counter, indicating how many valid elements are present in the buffer. If there is no valid element in this buffer, the buffer management sets up a flag to prevent the execution of any WD instruction. When new data is written into the WD buffer from the outside, the flag is reset.

- RD buffer: Controlling the RD buffer is more complex than other buffers. The number of valid bits in the RD buffer depends on the datatype size. The outside unit can provide a valid datatype size. Since there is an instruction to shift the buffer, the buffer management can explicitly deduce from the number of RDsh instructions that are executed when the buffer is empty.

6.4.4. WRITE VERIFICATION

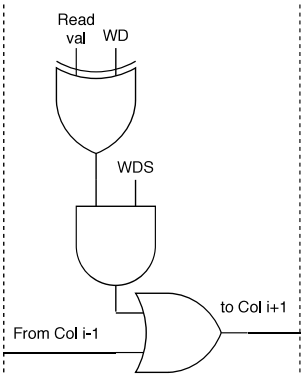


Figure 6.15: Implementation of write verification.

The write verification block only has to compare the crossbar output (stored in the read/logic register) to the data present in the WD register, setting the write-verify flag when one or more bits do not match. This comparison can be made by using a logical XOR gate. As only the output of the columns to which values were written should be verified, the comparison output is combined with the masking data from the WDS register through an AND gate. Finally, the logical OR operation can be used on the output produced for each of the columns to generate the verification flag, resulting in a logical '1' when a writing error has occurred. Figure 6.15 shows the implementation schematic per each crossbar column.

6.5. CIM TILE COMPILER

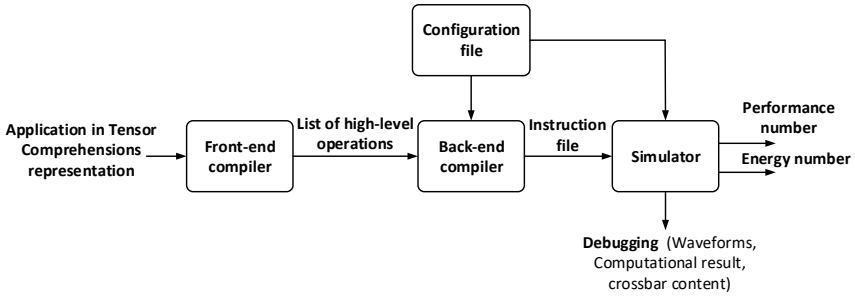


Figure 6.16: The overall system flow from tensor comprehensions down to fine-grained in-memory instructions

The compiler translates high-level operations intended for the CIM-Tile into a sequence of instructions to be executed within the tile. The high-level operations (e.g., MMM) are provided by the front-end compiler, which is responsible for searching for the operations within the application program that can be performed using the memristor crossbar (see Figure 6.16). The front-end compiler receives the application in *Tensor Comprehensions* representation. This is then converted to a polyhedral representation in order to identify computational patterns suitable for acceleration by employing *Loop Tactics*. The front-end compiler is written by our partners in MNEMOSENE project and more information can be found in [20, 222]. Based on the requirements or constraints that come from either the tile architecture or technology side, our back-end compiler translates high-level operations to in-memory instructions. As depicted in Figure 6.16, this information is written to the configuration file and passed to the compiler. For example, there might be a constraint on the number of rows that can be activated at once. This constraint can come either from the precision of ADCs or even technology capability. Therefore, if an operation wants to activate more rows, the compiler splits it into several steps and takes care of other changes that might be needed. The flexibility brought by our in-memory instructions helps to overcome constraints, requirements, and sparse patterns. Figure 6.8 illustrates an example for VMM operation where

four ADCs have to read columns in a special pattern (each 8 bit-lines share one ADC). This example demonstrates how the instructions deal with these kinds of patterns. It is important to note that the sequence of instructions generated by the compiler changes whenever the tile configuration changes. Therefore, by putting this complexity into the compiler, we try to keep the tile controller as simple as possible.

Table 6.6: Set of high-level operations and their semantics currently supported by the compiler.

Operation	Description	Semantic
MMM	Perform matrix-matrix multiplication between a matrix stored in the crossbar and an externally available matrix	MMM &A[d1][d2] - i - j - e - q - p - row - column
Store	Store a matrix at a specified location in the crossbar	Store &A[d1][d2] - i - j - p - q
Read	Read a matrix from a specified location from the crossbar	Read i - j - p - q
AND	Perform a logical AND operation on a specified set of rows and columns	AND v - j - q
OR	Performa logical OR operation on a specified set of rows and columns	OR v - j - q
XOR	Perform a logical XOR operation on a specified set of rows and columns	XOR v - j - q

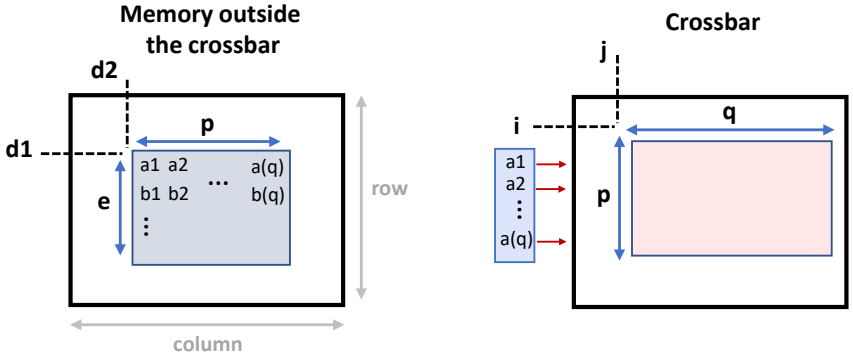


Figure 6.17: Illustration of parameters passed to the compiler for MMM operation.

Table 6.6 lists the high-level operations and their parameters passed to the (back-end) compiler. The input data provided to the crossbar comes from a memory where its address is embedded into the parameters. The implementation of this is left for the future, where multi-tiling and communication among tiles and host is supported. In the current version, the compiler creates two files for RD and WD buffers where the CIM-Tile receives its input from outside. The compiler fills these two files with the correct data according to the address provided in the parameters of high-level operations. In the future, this data should come from other tiles or external storage.

To clarify more on the parameters passed into the high-level operations, Figure 6.17 depicts an example of MMM operation. In this operation, we assume the multiplicand matrix is already stored in the crossbar by a 'store' operation. The data

of multiplayer is provided from memory outside the crossbar. The first parameter $\&A[d1][d2]$ provides the address to the first element of this matrix. Besides, 'i' and 'j' shows the coordination of the first element of multiplicand in the crossbar. 'P' represents the number of elements we have in one row of the multiplier, which should be equal to the number of elements in one column of the multiplicand. To clarify more, 'p' represents the number of active rows in the crossbar. It should be noted that the data from the multiplier matrix is provided to the crossbar row by row. Each row is copied to the RD buffer of the crossbar. We need to keep in mind each element may be represented by several bits. The RD buffer manages this for the crossbar. Finally, 'e' shows the number of rows in the multiplayer matrix, and 'q' represents the columns where the multiplicand matrix is stored. There are two other parameters that show the dimension of the external memory. This is required when we want to find the address of the first element in a new row in the multiplier matrix. However, for now, the compiler assumes a predefined size and ignores this information.

Similar to MMM, for the store operation, we identify the region in the crossbar where we want to write data by using 'i', 'j', 'p', and 'q' parameters. For now, we assume the data in the crossbar and outside memory are represented in the same way. We have to clarify that 'q' is the number of columns in the crossbar. If we store one bit per memristor and the datatype size is 8 bits, the number of elements we have in one row of crossbar or outside memory is $q/8$. Finally, for the logical operations, we want to activate two or more rows, and then, with a specialized sense amplifier, we compute the result of the operation. Based on what we observed from different applications, these activated rows (mainly two rows) are not necessarily next to each other and can be anywhere in the crossbar. In the current version, we pass a parameter 'v' to the compiler, which is a vector sized to the number of crossbar rows indicating which rows are selected.

6.6. CIM TILE SIMULATOR

The proposed CIM architecture is generalized, making it capable of targeting different technologies with different configurations of the peripheral circuit. The simulator, written in SystemC, models the architecture presented and generates performance and energy numbers by executing applications. The simulator takes as input the program generated by the compiler, which is currently stored as simple (human-readable) text. In our HDL implementation, they are translated into binary bits. Besides the program, to simplify design space exploration, the configuration of architecture has to be sent to the simulator via a configuration file in which the user is able to specify many parameters. The simulator produces as output the following: (1) energy and performance numbers, (2) content of the crossbar (over time), (3) waveforms of all control signals, and (4) the computational results. All outputs are written into text files to be used for further evaluation. Figure 6.18 illustrates the control and data flow of the simulator considering just 2-stage pipelining. By decoding an instruction, the data embedded in it along with the control signals, are passed to the corresponding component related to that specific instruction in

the data path. Then, the status will be returned to the controller to indicate the execution of the instruction is finished.

Table 6.7: List of parameters used in the configuration file

crossbar	drivers/analog peripheries	digital peripheries
- number of rows/columns - cell levels	- number of ADCs - precision of ADCs	- clock frequency - datatype size
- cell resistances - cell read/write voltages	- ADC power - read/write drivers power	- energy per adder in addition unit
- write latency - read latency	- ADC latency per conversion - SH latency	- RS/WD/WDS/CS filling cycle - instruction decoding cycle - latency per adder

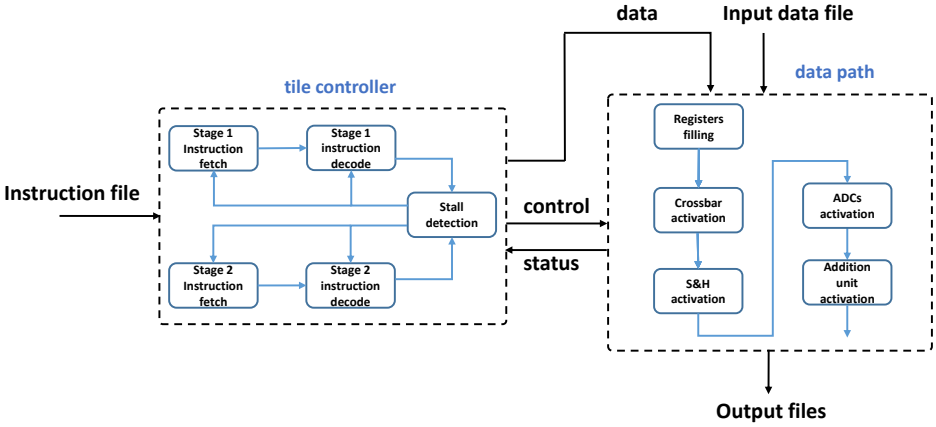


Figure 6.18: Simplified flowchart of the simulator comprising control flow and data path

The simulator has been written in a modular way, which helps us to easily modify or replace the components shown in the tile architecture with new designs/circuits. Moreover, new attributes can be easily added to the components model, and their impacts are captured at the kernel level (e.g., read/write variability for the crossbar). In this fully parameterized simulator, each component has its own characteristics like energy, latency, and precision written into the configuration file. Table 6.7 shows all the parameters that can be set in the file to be used for the early-stage design space exploration. Furthermore, the first-of-its-kind feature of our simulator is the ability to calculate energy numbers according to the data provided by the application. Existing simulators estimate average energy numbers regardless of data. Our simulator considers the data stored in the array to estimate the energy consumption in the crossbar and its drivers. This is achieved by taking into account the data stored in the crossbar cell resistance level, the number of activated rows,

and the equivalent resistance of the crossbar.

The power consumption of the crossbar and read drivers regarding read/compute operations is given in Equation 6.1 where R_{rc} is the resistance level of the memristor cell located in row "r" and column "c", $P_{DIM_{read}}$ is the read drivers power, and $V(read)$ is the read voltages. In general, R_{rc} and $V(read)$ are members of two sets containing possible resistance and voltage levels, respectively. Furthermore, $activation_r$ is a binary value that indicates whether row "r" is activated and contributes to the power of the crossbar or not. Considering read and compute operations, the summation is performed for the selected rows and all the columns. In addition, for simplicity, the resistance of access transistors, as well as bit-lines, are ignored. The power consumption of write operations is shown in Equation 6.2 where $P_{DIM_{write}}$ is the write drivers' power. $V(write)$ and $I(write)$ are the write voltage and programming current, respectively. In general, $V(write)$ is a member of a set including different write voltage levels. Finally, $activation_c$ determines whether the column "c" is activated and would contribute to the crossbar energy or not. The summation is performed over the selected columns in just one activated row. The energy consumption of read/computational as well write operations are shown in Equations 6.3 and 6.4, respectively, in which $T_{Xbar(write)}$ as well as $T_{Xbar(read)}$ are the latency of the crossbar for write and read/computational operations. The latency of the crossbar for read/computational operations also depends on the peripheral circuits used to capture or read the analog values generated by the crossbar (S&H). Using S&H unit to capture the result, its capacitance is charged with different gradient according to the equivalent resistance of the crossbar. Therefore, the result should be captured at the right time when there is a maximum voltage difference on the capacitance of S&H unit for different crossbar equivalent resistances, which helps to be distinguished by the ADC easily (implies that the crossbar latency also depends on ADC capability). It is worth mentioning that the equations are data-dependent and provide the worst-case energy numbers for the crossbar and its drivers.

$$P_{(read,compute)} = \sum_{r=1}^{\#rows} [(\sum_{c=1}^{\#columns} \frac{V^2(read)}{R_{rc}}) + P_{DIM_{read}}] * activation_r$$
$$R_{rc} \in \{L1, L2, ..., Ln\} \; activation_r \in \{0, 1\}$$
$$V(read) \in \{V(r1), V(r2), ..., V(rn)\} \tag{6.1}$$

$$P_{(write)_r} = \sum_{c=1}^{\#columns} (V(write) * I(write) * activation_c + P_{DIM_{write}})$$
$$V(write) \in \{V(w1), V(w2), ..., V(wn)\} \tag{6.2}$$

$$E_{(read,compute)} = P_{(read,compute)} * T_{Xbar(read,compute)} \tag{6.3}$$

$$E_{(write)} = P_{(write)} * T_{Xbar(write)} \tag{6.4}$$

6.7. FPGA VALIDATION

Before any simulation, the functionality of the design should be verified first. To this end, we have implemented the CIM-Tile on the Xilinx Zynq ZC702 board, where analog components are modeled with digital circuits. In this side experiment, the correct execution of the instructions and the generated final output were evaluated to validate the functionality of the architecture. Similar to the simulator, the HDL has been written in a parametrized way. This also allows for exploring and directly comparing the characteristics of different configurations of the design. The following parameters can be swept in the HDL design: 1) crossbar size, 2) maximum datatype size, 3) the number of available ADCs (its implication on the 'Addition Unit'), 4) block size for the block-wise masking register filling, and 5) bus bandwidth to communicate with outside. It should be noted that the compiler generates the program, and this is stored in the FPGA before the execution. In the following, we provide the initial results of our FPGA implementation.

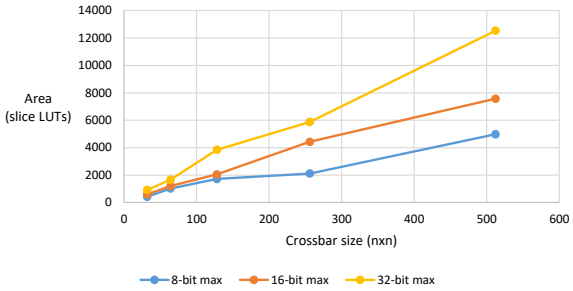


Figure 6.19: LUTs consumed by the digital CIM-Tile components.

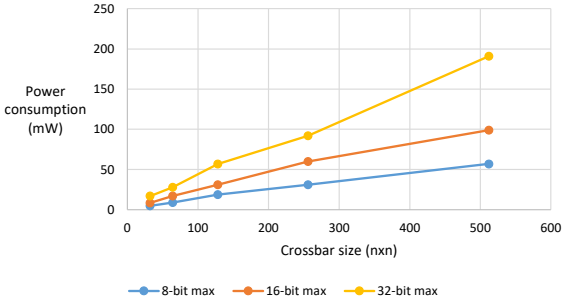


Figure 6.20: Power consumption of the digital CIM-Tile components implemented in FPGA.

In theory, most digital components of the CIM-tile architecture, such as the buffers, registers, and the addition unit, scale linearly with the crossbar size as long as a constant ratio between the crossbar size and the number of ADCs is

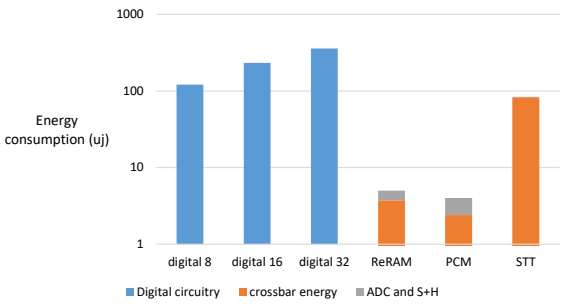


Figure 6.21: Energy comparison of Digital circuitry implemented in FPGA and analog components in 90 MHz clock frequency.

maintained. Figure 6.19 and 6.20 show the LUTs and power consumption reported by Vivado. In these reports, we only included the resources/power consumed by the digital parts of the CIM-Tile (excluding the crossbar model, ADC, S&H, and Drivers). The simulation is done for three different datatype sizes, showing almost a linear increase in area usage and power consumption for each as we increase the crossbar size. In addition, the effect of supporting different datatypes can be seen. Clearly, as we increase the datatype size, more area and larger power are consumed due to mainly a larger RD buffer as well as an 'Addition Unit'.

Using the simulator, accurate numbers for energy consumption, taking actual data, and analog component activation into account can be obtained. The 'GEMM' benchmark is used to obtain these numbers. Figure 6.21 compares the energy consumption of the analog components and the digital circuitry for executing the benchmark at a clock frequency of 90MHz. It can be seen that the energy consumption for running the benchmark is significantly higher. This is because the analog components are not activated during the entire program. It should be kept in mind that the presented power and energy figures of the digital circuitry only represent the actual FPGA implementation of the circuitry. An ASIC implementation of the circuitry shows way lower power consumption, as seen in the following section.

6.8. ASIC EVALUATION

Despite the previous section, we evaluate the ASIC implementation of the proposed architecture in terms of power, energy, and area compared with analog components. As a benchmark, the linear-algebra kernel "GEMM" from the Polybench benchmark suite was chosen. In this kernel, first, the multiplicands are written into the crossbar (write operation), and then the actual multiplication is performed. The values regarding the CIM-tile analog components are summarized in Table 6.8. The parameters for ReRAM and PCM technologies are taken from [223] and [82] validated for actual devices. The same ADC values used in [67] are considered for our setups. To obtain the power consumption and area for the CIM-tile digital circuits, they

were synthesized in Cadence Genus targeting standard cell 15 nm Nangate library. Since all the information and control signals can be tracked using our simulator, a typical **activity factor** and performance number are extracted. These numbers are incorporated to obtain accurate energy consumption for the digital as well as analog components of the tile.

Table 6.8: Value of parameters used for the analog components.

Component	Parameters	Spec	
Memristive devices		ReRAM	PCM
	cell levels	2	2
	LRS	5k	20k
	HRS	1M	10M
	read voltage	0.2V	0.2V
	write voltage	2V	1V
	write current	100 μA	300 μA
	read time	10 ns	10 ns
	write time	100 ns	100 ns
Crossbar	structure	1T1R	
	num. columns	256	
	num. rows	256	
S&H	number	256	
	hold time	9.2 ms	
	latching energy	0.25 pJ	
	latency	0.6 ns	
DIM		read DIM	write DIM
	number	256	256
	power	3.9 μW	3.9 μW
ADC	power	2.6 mW	
	precision	8 bits	
	latency	1.2 GSps	

As we mentioned before, the energy consumption of the crossbar depends on the input and programming data. As more devices are programmed to LRS, and more rows are activated, clearly more energy is consumed during the computation. Since the simulator can actually execute kernels, the energy number obtained for the crossbar is data-dependent. Figure 6.22 depicts the energy consumption of the crossbar with different levels of sparsity. This figure illustrates the sparsity of logic value 1 for matrix-matrix multiplication. For this experiment, it is considered that the multiplier (input) and the multiplicand (programmed) matrix have the same sparsity. In addition, logic value 1 (0) as an input implies the corresponding row is activated (deactivated) and as programming data indicates, the memristive device is programmed to LRS (HRS). The simulation is performed for ReRAM and PCM devices, and an interesting observation is that although the ratio of HRS to LRS is lower in ReRAM devices, since PCM devices have higher LRS compared to ReRAM, the sparsity leads to less variation in the energy consumption of the crossbar made with PCM.

Considering our CIM-tile architecture, digital circuits have different contributions to power consumption. Figure 6.23, depicted for 8-bit maximum supported datatype size in the tile, gives an insight into how much power each of these circuits consumes (8-bit datatype size means that in the case of MMM, each element of the multiplicand is distributed over 8 cells and each element of the multiplier is

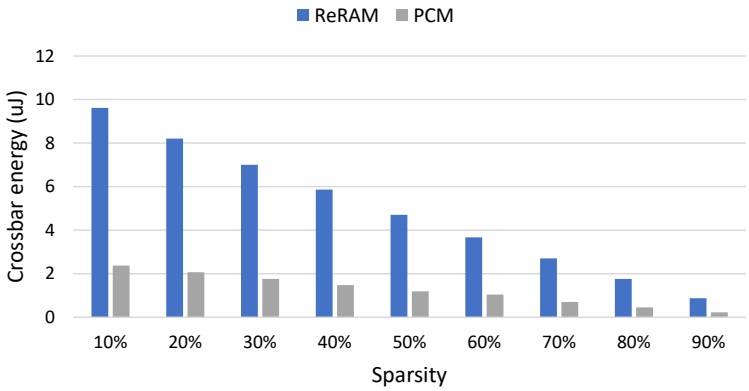


Figure 6.22: Energy consumption of the crossbar with respect to the percentage of sparsity for both input and programming data.

fed to the crossbar over 8 steps). As we expected, the *RD* buffer due to its size (feeding 8-bit data to 256 crossbar rows) and *addition unit* due to its high number of instances consume more power than others. However, since the buffer is not always switching (dynamic power), the average power is much less. The imposed power consumption on the system is the cost of the flexibility of row selection brought by the *RD* buffer and *RDS* register and their associate instructions.

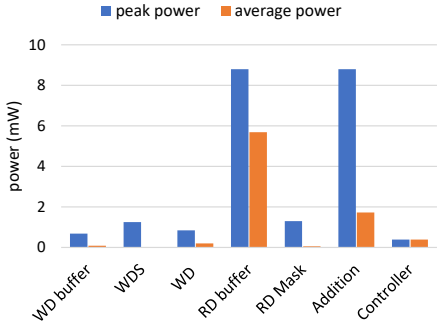


Figure 6.23: Power breakdown of digital circuits.

The overhead of digital circuits in terms of energy is depicted in Figure 6.24 (a) and (b) to quantify the expense of high flexibility for programming the tile. In this figure, the overhead of the digital parts of our tile is compared with the analog parts. The comparison was performed for both ReRAM and PCM technologies with 256*256 crossbar size considering different datatype sizes. **First**, it is observed that as the datatype size increases, due to more computations, the overall energy is increasing as well. **Second**, by increasing the number of computations, the overhead

of crossbar **pr**ogramming reduces (see striped blue bar) since more computations are performed before reprogramming the crossbar. **Third**, digital circuits impose more overhead while moving from 16-bit to 32-bit datatype size (see orange bars as well as energy per VMM). As the datatype size increases, fewer adders and registers must be instantiated (fewer numbers produced per crossbar activation). These units are essential to accomplish digital post-processing on the crossbar's output to deliver the final result. However, as the data type size increased, each of these adders and registers individually is of a larger size (see 7). Accordingly, although this customized unit utilizes minimum-size adders and registers, supporting larger datatype size resulted in more overhead for this unit. **Fourth**, the overhead of the digital circuit for the PCM case is higher compared to ReRAM since the PCM crossbar consumes less energy due to its higher value of low resistance state. While the device level researchers are working on devices with a higher value of low resistance, this graph gives a good insight into how much this value contributes to the energy consumption of the system.

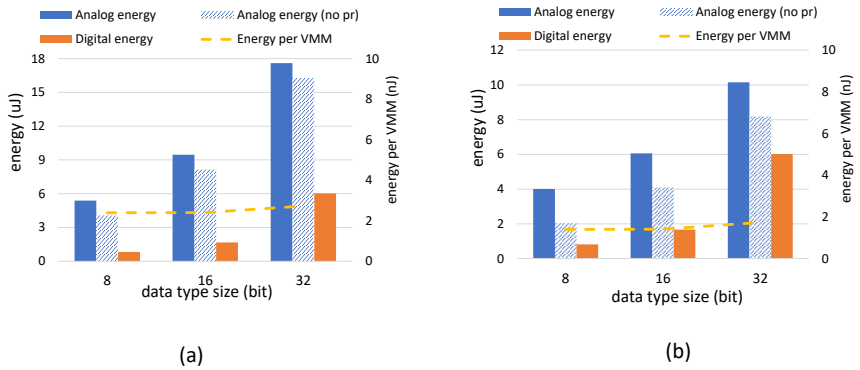


Figure 6.24: (a) Energy number for different datatype sizes considering ReRAM and (b) PCM with crossbar size of 256×256. It is assumed the entire crossbar contributes to the computation.

Finally, the area comparison of digital and analog circuits is depicted in Figure 6.25. For this experiment, the area of each cell in 1T1R crossbar structure is taken from [178], where it was fabricated with 22 nm technology. Although our digital circuits were synthesized with 15 nm technology, the result shows their area is around 6 times less than the analog counterpart, regardless of the crossbar dimension.

6.9. CONCLUSION

In this chapter, we presented our programmable CIM-tile architecture for which, by exploiting our ISA, high flexibility is achieved. In addition, the architecture provides a clear interface and independence from external devices. Considering different CIM-tile configurations, we developed our compiler and simulator to facilitate design space exploration. By synthesizing the digital part of the CIM-tile, its overhead

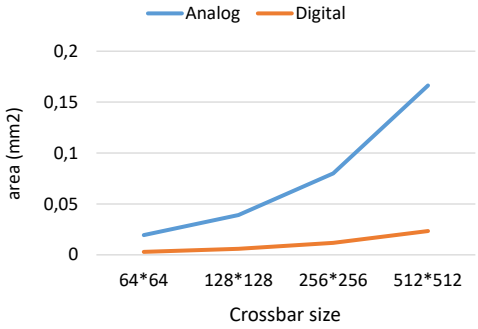


Figure 6.25: Area comparison of digital/analog circuits for ReRAM.

compared to the analog counterpart was demonstrated in terms of power, energy, and area. The result will give a remarkable insight into future research to realize which customizations have to be applied for different configurations and application requirements. In addition, in the future, the proposed CIM-Tile should be compared with advanced digital counterparts that provide a similar level of flexibility (e.g., GPUs or TPUs).

7

EFFICIENT DIGITAL PERIPHERY DESIGN FOR CIM-TILE

Another critical component in the CIM-Tile is the 'Addition Unit', where we perform additional processing on the memristor crossbar output. We enable signed and unsigned arithmetic operations in the CIM-Tile by employing our proposed energy and area-efficient 'Addition Unit'. Our approach combines a) the mapping of the signed operands on the 1T1R crossbar, and b) the augmentation of the periphery with customized circuits to support the execution of shift and accumulate needed for the arithmetic operations. The operand mapping is performed without the need for sign extension; hence, reducing the required memory size. This unit is controlled with our (micro) ISA discussed in the previous chapter. The flexibility of the design allows applications to dynamically switch between signed and unsigned computation as well as different datatype sizes without changing the hardware or mapping of data. This is another step toward a generic CIM-Tile design.

This chapter is based on [26], [27], and [220].

7.1. DIGITAL PERIPHERY DESIGN CHALLENGE

In this chapter, we are focusing on a critical component, 'Addition Unit', in the CIM-tile organization. Figure 7.1 illustrates the focus of this chapter in our generic image of the tile. As we discussed, performing the computation in memory using a memristor device can reduce the cost of communication in traditional computers. However, inefficient periphery circuits can completely alleviate the gains. As an example, many applications such as Neural Networks (NNs) and Convolutional Neural Networks (CNN) at least require integer (number) computations. However, since only limited levels can be stored in one memristor cell, the (bit-)vector representing the number should be distributed over multiple cells, which requires some extra processing outside the crossbar to get a meaningful result. Therefore, great care must be taken in the design of these peripheral circuits to really deliver efficient in-memory computing systems. Besides, from the application perspective, mapping the operands to the memory unit, comprising emerging devices and their peripheries, is a critical step toward enabling energy-efficient CIM; it includes choosing the appropriate datatype structure and supporting signed numbers. These applications are often operating on signed numbers and require different data type sizes. Hence, without the support of signed numbers, the promise of CIM for these applications is greatly diminished. Therefore, it is essential for researchers to enable this feature for CIM-based design as well. Since memristor devices store the information as different (positive) conductance levels, additional considerations are required when the computation has to be performed on signed numbers. This enforces more complexity in the way the data is mapped to the crossbar as well as the way the computation should be performed in the periphery. Therefore, simple yet energy-efficient digital periphery are needed to perform operations on both unsigned and signed numbers.

7.2. CONTRIBUTION

In this chapter, we propose a novel mapping solution to support signed and unsigned MMM based on widely used two's complement representation. In this method, signed and unsigned operations are performed with minimum energy, latency, and area overhead by eliminating the costly sign extension. The proposed periphery architecture utilizes minimum-sized adders customized based on technology-driven restrictions. The structure is not restricted to a single fixed integer word size and can flexibly support different word sizes without hardware changes. In short, this chapter presents the following main contributions:

1. *A novel scheme to support widely used two's complement arithmetic operations*: This is accomplished by exploiting the behavior of the memristor crossbar without the costly sign extension. This eventuates to efficient support of signed integer/fix-point computations in terms of energy, latency, and area. The flexibility of the design allows applications to dynamically switch between signed and unsigned computation without changing the hardware or mapping of data.

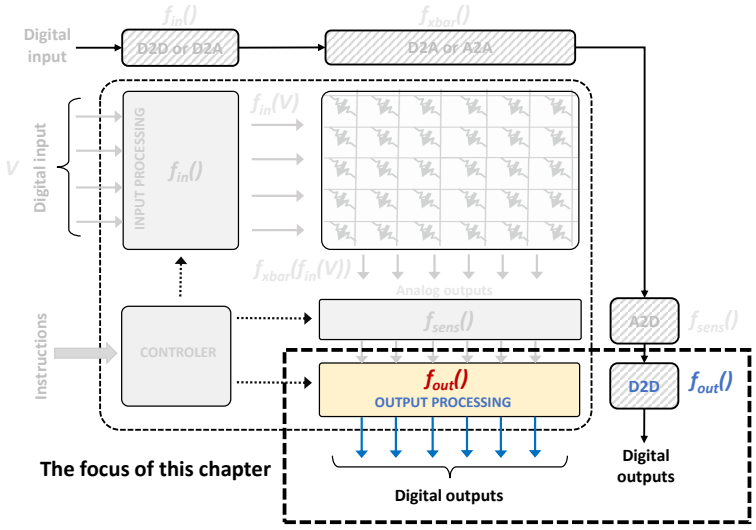


Figure 7.1: We focus on the 'Addition Unit' in this chapter, where the crossbar output is processed in a way to support unsigned and signed integer computing. This part is highlighted in the figure.

2. *An energy and area efficient digital periphery architecture for MMM*: The periphery fulfills additional processing required for MMM using minimum-sized adders and registers. The proposed solution allows applications to dynamically change the data size without changing the hardware or the mapping scheme of data to the crossbar.

3. *Evaluation of the proposed solution using two memristor technologies (RRAM and PCM) and different benchmarks*: The design is validated and evaluated in terms of energy (for both computation and programming phases), execution time, and area while considering RRAM and PCM technologies.

7.3. STATE-OF-THE-ART DESIGNS

There are limited works focusing on data representation and periphery design for the CIM crossbars. Supporting unsigned integer numbers is the fundamental step to be able to execute limitted applications. Since the memristor devices can hold limited bits, traditionally, the unsigned integer data is distributed over several memristor devices. This distribution can be balanced [224] or unbalanced [15] if we want to allocate more devices to the most significant bits. The periphery is assumed to be a simple 'shift and Add' unit to integrate all intermediate results and produce the final output. We discuss this in detail in Section 7.4. To the best of our knowledge, CASCADE [225] is the only work that took a different approach and also provided implementation details to support integer numbers for MMM. In this work, by

employing analog buffers, some parts of partial sum and accumulation happen in an analog way, which helps to reduce the number of ADC conversions. Although the number of conversions is decreased by leveraging this approach, the resolution of ADCs should be increased, which imposes energy overhead that grows exponentially [67]. Furthermore, as the number of columns contribute to the analog partial sum and accumulation increased, the relative difference between voltage/current levels of output reduced, which requires more complex ADCs. Equations 7.1 and 7.2 demonstrate a set of equivalent resistance for one column as well as several columns when they are attached to each other, respectively.

$$R_{eq} \in L = \{R_{off}, R_{on}, R_{on}/2, ..., R_{on}/n\} \quad n: number\ of\ crossbar\ rows \tag{7.1}$$

$$R_{eq} \in L = \{R_{off}, R_{on}, R_{on}/2, ..., R_{on}/(m*n)\} \quad \begin{matrix} n: number\ of\ crossbar\ rows \\ m: number\ of\ connected\ columns \end{matrix} \tag{7.2}$$

To enable CIM even for more applications, we need to support signed representation for both (1) the data programmed into the crossbar –***array data***– and (2) the data provided as input to the crossbar –***input data***. However, to the best of our knowledge, there are only a few works that partially look into this problem and propose some solutions. In the following, we discuss the existing solutions on how they represent signed numbers for both *array data* and *input data*.

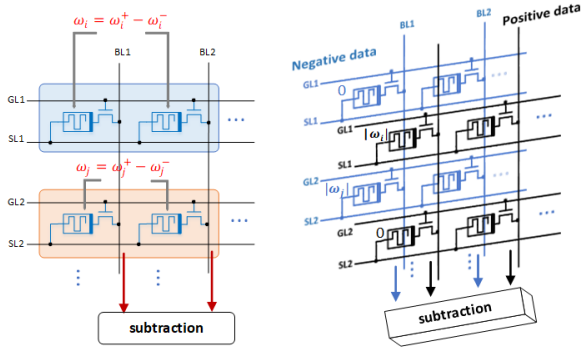


Figure 7.2: Supporting signed array data by (a) subtraction of cells' value (b) mapping negative and positive values to different crossbars.

7.3.1. SIGNED NUMBER REPRESENTATION OF ARRAY DATA

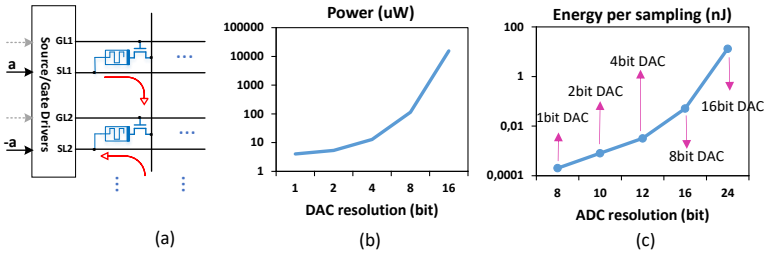
In order to support negative numbers, ISAAC [67] brings the array data into a positive range by using a bias value. This imposes two types of extra processing. First, additional pre-processing if the data has to be programmed to the crossbar on the fly to bring them into the positive range. Second, mandatory post-processing

to subtract the bias value several times equal to the number of array data that are added up. This has to be performed for every output element. Another approach represents a number as a difference of two or more cells [226, 227]. Figure 7.2(a) illustrates this approach assuming that the data is presented as a subtraction of two numbers, each residing in a single cell. PRIME [110] and PipeLayer [69] map positive and negative numbers into different crossbars (as unsigned numbers). Figure 7.2(b) depicts how a matrix with positive and negative numbers is mapped to two crossbars. In both cases, the result is obtained based on subtraction. Besides having more cells to represent a number, which leads to more energy/area consumption, more complexity is imposed on mapping data to the crossbars, especially when it is required to incorporate more cells for large datatype sizes. Another approach is using two's complement representation [65]. However, due to the necessity of sign extension, a big overhead is imposed in terms of area and energy (more details are in Section 7.5). Finally, FloatPIM [228] uses the IEEE standard floating-point to store data as a floating-point number in the crossbar. Using a sign bit to indicate the sign of a number, as well as the way of computation in the crossbar, limits this approach to perform multiplication and addition between only two numbers. This inherently differs from fulfilling vector-matrix-multiplication (and, in turn, MMM) at once by exploiting Kirchhoff and Ohm's laws.

7.3.2. SIGNED NUMBER REPRESENTATION OF INPUT DATA

A few works have studied how the input data to the crossbar can be represented. ISAAC [67] provides 16-bit data to the crossbar in 16 cycles (one bit per cycle) in 2's complement format. As mentioned for array data, a big overhead is imposed due to the sign extension. Another approach is to reverse the direction of the current for negative input as depicted in Figure 7.3(a); this is only employed for subtract operation in [65]. Considering this approach, the analog input voltage levels that have to be supported by the Digital-to-Analog Converters (DACs) should be doubled. Consequently, more voltage/current levels should be distinguished by ADCs. Therefore, not only the system becomes more sensitive and error-prone, but also its power consumption/energy increase exponentially as the resolution of DACs and ADCs increase. This is shown in Figure 7.3(b) and 7.3(c), presenting the power and energy per sample for a DAC and an ADC, respectively, by increasing the resolution [193, 194]. It should be noted that the minimum required ADC resolution is based on the DAC resolution and the number of array data contributing to the analog addition (in this case, 256). As an example, if we have 2-bit DAC, the ADC resolution is calculated as $log2(256) + 2 = 10$ bits.

In conclusion, considering the limitations and overhead of existing solutions to support sign and unsigned arithmetic computation in the crossbar, it is essential to reduce this overhead with an efficient crossbar periphery architecture. Any architectural solution for the digital periphery should take into account the following restrictions as well: 1) ADC resolution; this limits the maximum number of crossbar rows to be activated. 2) Limited resistance levels on a memristor device; if a number cannot be represented by these few levels in a single device, it has to be distributed over several devices. 3) Input drivers (DACs) resolution; due to the limitation on



(a) (b) (c)

Figure 7.3: (a) Bi-directional input current to support signed input data and its implication on power/energy consumption of (b) DACs and (c) ADCs in different resolutions (assuming 256 × 256 crossbar with 1-bit per cell).

the number of voltage/current levels provided by the drivers to the crossbar, input data has to be split and fed to the crossbar in several steps. Considering the aforementioned limitations, more processing steps have to be performed on the output of the crossbar to achieve the final result for arithmetic operations.

7.4. EFFICIENT ARITHMETIC COMPUTATION IN CIM TILE FOR UNSIGNED DATA REPRESENTATION

In this section, we propose an efficient structure for the digital periphery required next to a memristor crossbar to execute a complete unsigned addition and MMM operations. We first focus on unsigned data representation and computation. The presented structure forms the basis for supporting signed computations, which we will describe in Section 7.5.

Our proposed design consists of three stages: 1) Analog Addition; this stage addresses the first limitation mentioned before (ADC resolution). 2) Sliding over multiple columns; the hardware designed for this stage addresses the second limitation (limited memristor resistance levels) 3) Sliding over input segments; this stage takes into account the third limitation (DACs resolution). These three stages are executed consecutively for several iterations in order to produce a final result for arithmetic operations. We explain each stage in the following subsections. The proposed design utilizes minimum size adders to lessen the energy and latency overhead of periphery circuits. Our approach is flexible and can be applied to a system with a different configuration (e.g., ADC/DAC resolution, resistance levels, datatype size).

Figure 7.4(a) depicts an example of a MMM operation where each element is assumed to have three bits. Considering this operation, the elements of the multiplicand matrix (array data) have to be programmed into the crossbar while the multiplier's elements (input data) are provided to the crossbar as input. Figure 7.4(b) depicts how an element of the output matrix can be calculated and mapped to the crossbar, assuming 1) elements of the multiplicand matrix have to be distributed over three memristor cells (due to the restriction on the number of resistance levels

– second limitation) and 2) the elements of multiplier matrix have to be sliced into three segments as well (due to the restriction on the number of voltage/current levels provided by DACs – third limitation). The segments are given to the DACs sequentially (e.g., the first segment is a_0, b_0, and c_0).

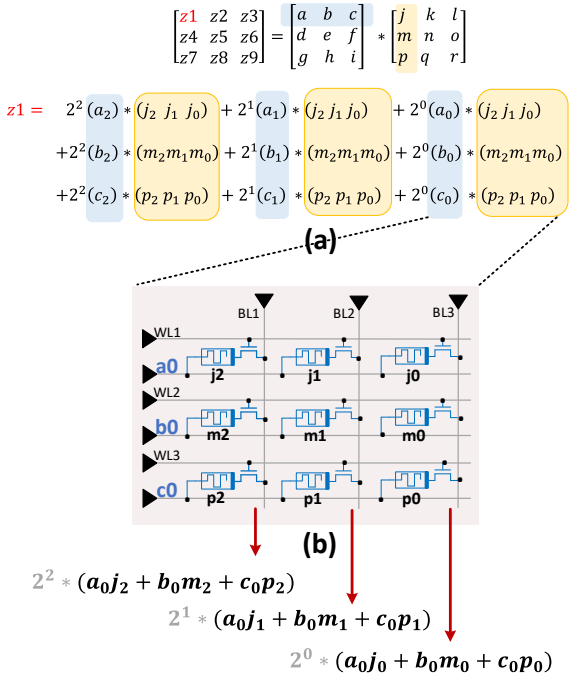


Figure 7.4: (a) An example of MMM operation where the elements of the array and input data are expanded (b) mapping of data to the crossbar considering limited resistance levels as well as limited input voltage/current levels. The coefficients colored gray are not part of the crossbar output.

7.4.1. FIRST STAGE: ANALOG ADDITION

The first stage of computation is performed in an analog manner. As depicted in Figure 7.4(b), elements with the same significant bit are summed up. The analog addition is performed on the bit-lines as soon as the DACs are activated and proper voltage levels are provided to the crossbar inputs. In an ideal case, all the crossbar rows can be activated. Therefore, all the required elements can contribute to the analog addition. However, due to some limitations, this may not be realized.

Figure 7.5 illustrates a scenario where all the input drivers (DACs) cannot be activated at the same time. This can be due to either technology restrictions or limited ADC resolution (first limitation). Hence, the analog addition should be performed among smaller sets of rows, and the intermediate results need to be

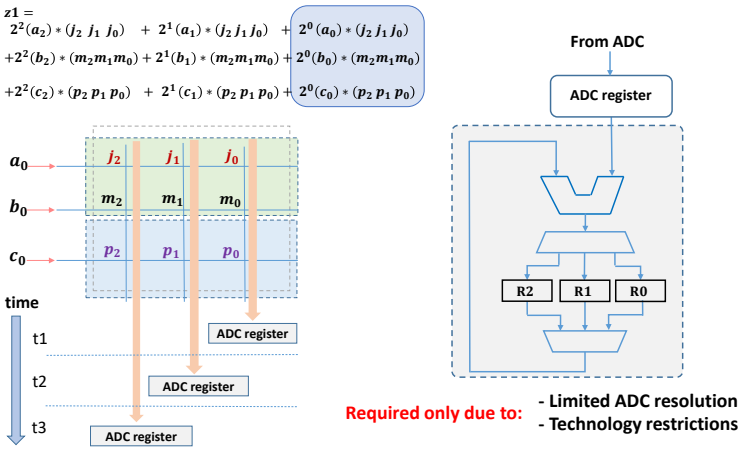


Figure 7.5: First stage: Activating rows in multi-steps and its required hardware when there is a limitation on ADC resolution or crossbar technology.

summed up together in the digital domain. In the example illustrated in this figure, the first two rows (green box) are activated in the first step. Afterward, the bit-lines are scanned by the ADC in sequence, and the intermediate results are stored in the registers dedicated for each column ($R1, R2, R3$). In the second step, the third row (blue box) gets activated. The results in this step have to be accumulated with the previous step. We present the required hardware to deal with this restriction in Figure 7.5. Based on which bit-line is read by the ADC, control signals are sent to the demultiplexer and multiplexer to load and store data from and into a proper register, respectively. This hardware is only required when this limitation exists. The size of the registers employed for each column should be equal to $log2(number\ of\ rows) + log2(resistance\ levels)$. The second term in the equation is added when the number of resistance levels is more than two.

7.4.2. SECOND STAGE: SLIDING OVER MULTIPLE COLUMNS

The intermediate results obtained from each column of the crossbar in the first stage just contain one bit-position of array data (it can be more if memristor devices can hold more than one bit). In order to achieve the final results related to one segment of the input data (e.g., $a_0, b_0,\ and\ c_0$), the intermediate results of the first stage obtained from different columns, have to be summed up (e.g., column 1, 2, and 3). This is due to deploying several memristor devices to represent a number (second limitation). In the following, we explain the traditional and proposed approach to perform the required processes in this stage.

As depicted in Figure 7.6, in each time step, the result of analog addition obtained from a column of the crossbar is stored in the *ADC register*. In the traditional approach, the data stored in this register has to be shifted to take into account the coefficients associated with each column (see Figure 4.3(b)). Then, the result is

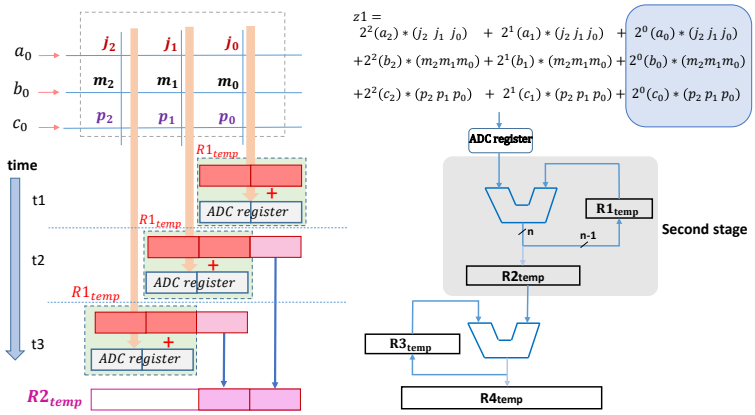


Figure 7.6: Second stage: accumulation of partial result obtained from columns representing a number. The size of adders and registers are minimized in this stage.

accumulated to the values obtained from other columns in the previous time steps. Considering that approach, the number of shift operations is variable and depends on to which column the value of the ADC register belongs. In addition, as more columns contribute to the result, the data size gets increased. Therefore, the adder which performs this accumulation should be sized for the worst-case scenario. This size is equal to $log2(number\ of\ rows) + log2(resistance\ levels) + collective\ columns$. The collective columns are the number of columns that represents a single value (e.g., in this example, 3). In short, this approach requires a variable number of shift operations and large-size adders, which in consequence, reduce the performance and energy efficiency of the crossbar periphery.

In contrast to the traditional approach, our proposed design does not require shift operation and minimizes the size of adders as well as registers that are placed in this stage. Considering Figure 7.6, the value of the ADC register has to be shifted and added up to the value of $R1_{temp}$ register where the intermediate result obtained from the previous columns is stored. In the proposed design, when performing addition between these two registers, the least significant bit (or bits when more than one bit is stored in a single memristor cell) can be directly sent to the next level register ($R1_{temp}$) since it will not contribute to the later additions. As an example, in the first time step, the value of the ADC register is added to the value of $R1_{temp}$ (initialized to 0) and stored again in $R1_{temp}$. In the second time step, since the value of the ADC register, which currently stores the value of the second column, has to be shifted once to the left, the least significant bit of $R1_{temp}$ does not contribute to the addition. Therefore, it can be directly passed to the $R1_{temp}$ where we store the final result of this stage. As a consequence, the shift operation is implicitly implemented in the design without dedicating extra hardware to it. In addition, the size of $R1_{temp}$ register and the adder module are held as minimum

as possible equal to $log2(number\ of\ rows) + log2(resistance\ levels)$. When the addition is performed for the last column (associated with the most significant bit(s) of multiplicand's elements), the entire content of $R1_{temp}$ register is copied to $R2_{temp}$ register, and the result of this stage is ready to be used for the next stage.

We perform the processing in this stage, assuming that the datatype size for array data (e.g., j_0, J_1, and J_2) does not exceed the number of columns that share an ADC. Loosening this assumption leads to additional logic (extra stage) required to add up the intermediate results obtained from the columns representing one element of the multiplicand, but are shared among more than one ADC. To avoid complexity, this will not be discussed in detail.

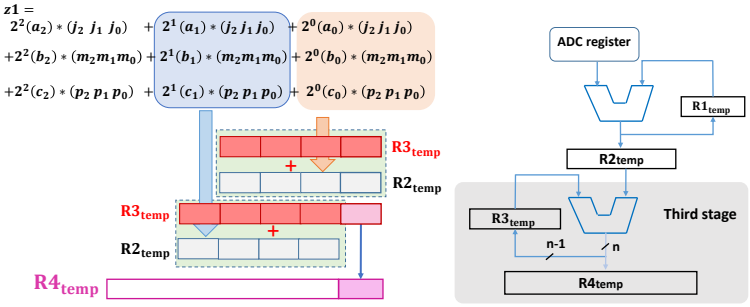


Figure 7.7: Third stage: performing addition between the higher partial results which are obtained from the input segments representing a number. The same technique applied in the second stage is used here to optimize the hardware.

7

7.4.3. THIRD STAGE: SLIDING OVER INPUT SEGMENTS

In this stage, the partial sum related to input segments has to be summed. Due to the input driver resolution (third limitation), the input data has to be segmented and provided to the crossbar in several iterations. Therefore, the design should be able to integrate the result of each iteration and produce the output of the MMM operation.

We illustrate this using an example in Figure 7.7 where the partial sum relating to the (a_0, b_0, c_0), (a_1, b_1, c_1), and (a_2, b_2, c_2) segments have to be integrated. The partial sum obtained from each input segment has to be shifted and added up to the value of $R3_{temp}$ register, where we store the result related to previous input segments. The design utilizes the same approach, deployed for the second stage, in order to avoid shift operations while minimizing the size of adders and registers. As an example, in the first iteration, the value of the first input segment (orange box) is available in $R2_{temp}$ register and will be added to the value of $R3_{temp}$ register, which is initialized to zero. The least significant bit of the result is directly stored in $R4_{temp}$ register, while the rest of the bits are again stored in $R3_{temp}$ register. Since the result of the second input segment (blue box) has to be shifted (theoretically) once to the left, the least significant bit of the result from the previous iteration is not contributing

to the result of the current iteration. Therefore, we improve performance, area, and energy efficiency by omitting variable shift operators as well as minimizing adder and register sizes.

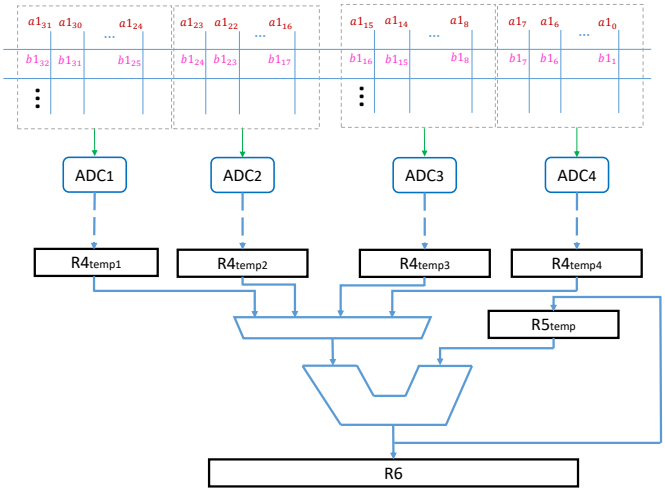


Figure 7.8: The possible organization required between ADCs.

What we discussed so far was based on the assumption that the size of integer numbers stored in the crossbar matches the number of columns shared by an ADC. However, two other possible scenarios have to be considered. First, by increasing the number of ADCs, a number stored in the crossbar might be split and distributed over multiple ADCs. To clarify this further, Figure 7.8 illustrates this scenario in which 32-bit numbers are distributed over 4 ADCs. Accordingly, to obtain the final result, the values stored in all the $R4_{temp}$ registers, employed for each ADC, have to be summed up. Although more hardware is required as the number of ADCs increases, the length of the registers and adders employed for processes per ADC are decreased. The second scenario is when two or more sets of data (due to a small datatype size) can be distributed and stored in the columns shared by one ADC. Considering this scenario, the first solution is replicating all the registers in the 'Addition Unit' to store the intermediate results per set of data that share an ADC. However, since we need to be prepared for the worst-case scenario (smallest datatype size), the overhead of extra registers would be considerable. The second solution is computing the (final) result sequentially for each set of data placed in the same columns. As an example, if we can store two elements residing in the columns that share an ADC, we first finalize the computation for the first half. Then, we start the computation for the second half of the columns. In our implementation, we follow the second solution.

In summary, using the aforementioned organization leads to energy (and possibly performance) improvement specially when considering a large number of crossbars with hundreds of columns computing in parallel. Controlling this structure can be

performed by designing a state machine. Alternatively, the entire system can also be controlled by employing an instruction set to exploit the maximum flexibility for the design (considering ADC resolution, cell levels, and datatype size).

7.4.4. INSTRUCTIONS RELATED TO THE ADDITION UNIT

Similar to other components in the CIM-tile, the 'Addition Unit' also has its own instruction to control this unit and provide flexibility in supporting different configurations. Table 7.1 highlights the instructions associated with this unit. In the following, we explain each instruction:

Table 7.1: The highlighted instructions are used for the 'Addition Unit'

Opcode	Op 1	Op 2	Function description
RDSb	Index	Mask	Place 'Mask' into RDS reg at 'Index'
RDSc			Clear the entire RDS register
RDSs			Set the entire RDS register
RDsh			Shift RD buffer contents
WDb	Index		Copy data to for WD buffer to WD reg. at 'Index'
WDSb	Index	Mask	Place 'Mask' into WDS reg at 'Index'
WDSc			Clear the entire WDS register
WDSs			Set the entire WDS register
FS	Function		Select crossbar functionality
DoA			Activate the crossbar function
DoS			Sample the crossbar output
CS	Index	Activation	Select column to be read by ADC
DoR			Activate the ADC
jal	Address		Jump to 'Address' and store PC.
jr			Jump to the PC stored in return reg.
BNE			Branch to PC stored in branch reg.
LS			Indicate the last section of rows to reads
IADD			Activate third stage of the addition unit
CP			Copy result per ADC to output buffer
AS	Selection		Select adders for addition between ADCs
CB			Copy the result of addition between ADC to output buffer

- **LS**: This instruction controls the first stage of the addition unit (see Figure 7.5). The instruction is defined to indicate whether this is the Last Section (LS) of row selection for this stage or not. In case this is the last iteration, the result in each of the registers considered for this stage (e.g., $R0, R1, R2$) has to be added with $R1_{temp}$ in the second stage.
- **IADD**: This instruction is defined to demonstrate that this is the last bit or column that we are processing for the second (or Intermediate) stage of ADDition (IADD). For example, if the datatype size is 8-bit, this instruction is placed after reading eight columns (assuming memristors in each column hold one bit). This is also illustrated in Figure 7.9, where a sample code was compiled for 8-bit datatype size. In this code, after eight 'CSR' (CS + DoR)

instructions, there is one 'IADD' instruction. After executing this instruction, the content of $R1_{temp}$ is copied to $R2_{temp}$.

- **CP**: Similar to IADD, CP shows that the last bit of input data was given to the crossbar (e.g., a_2, b_2, c_2. By executing this instruction, the content of $R3_{temp}$ is copied to $R4_{temp}$, and the final result of the third stage will be ready.

- **AS**: This instruction was defined to support an addition between the final results generated per different ADCs. This instruction controls the multiplexer illustrated in Figure 7.8 to select proper registers. This helps us to support datatype sizes lower than the maximum datatype size that the system is designed for. It should be noted that this instruction and its associated hardware are only used when data has to be distributed over the columns assigned to different ADCs. In case the maximum datatype size can fit within the columns shared by an ADC, this hardware is not placed, and the instruction is not used anymore.

- **CB**: Similar to CP, this instruction indicates that the last iteration for the addition between the $R4_{temp}$ registers assigned to different ADCs is performed. By executing this instruction, the content of $R5_{temp}$ is copied to $R6$ register (see Figure 7.8).

```
1.  FS      VMM
2.  RDsh
3.  RDSs
4.  DoA
5.  DoS
6.  LS
7.  CSR     000   11111111111111111111111111111111
8.  CSR     001   11111111111111111111111111111111
9.  CSR     010   11111111111111111111111111111111
10. CSR     011   11111111111111111111111111111111
11. CSR     100   11111111111111111111111111111111
12. CSR     101   11111111111111111111111111111111
13. CSR     110   11111111111111111111111111111111
14. CSR     111   11111111111111111111111111111111
15. jr
16. IADD
17. ...
18. jal     7
19. IADD
20. CP
```

Figure 7.9: An example of instruction sequence generated by the compiler with highlighted instructions related to the 'Addition Unit'.

7.4.5. IMPLEMENTATION CHALLENGE

As mentioned before, the least significant bit can be directly sent to the next level register. However, the hardware implementation for positioning the LSB in the right place in the next-level register might be challenging. Since the number of required shifts depends on the datatype size, a regular shift register cannot address this issue. As an example, assume the system is designed for 32-bit data with a crossbar size of 256×256. Then, the $R2_{temp}$ size would be $8+32=40$ bits. However, if we perform the computation on an 8-bit datatype size, the LSB is shifted only seven times, meaning that the result will be stored in position 39 down to 23 instead of position 15 down to 0 in the register. The same problem can happen for $R4_{temp}$. In the following we present three solutions for this [220]:

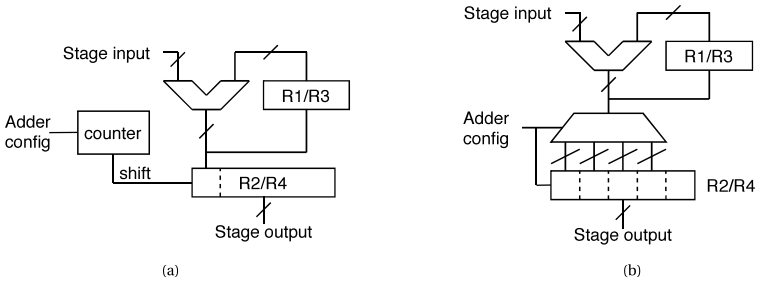


Figure 7.10: Implementation solution for positioning the LSB by using (a) a counter or (b) multiplexer to support flexible starting point [220].

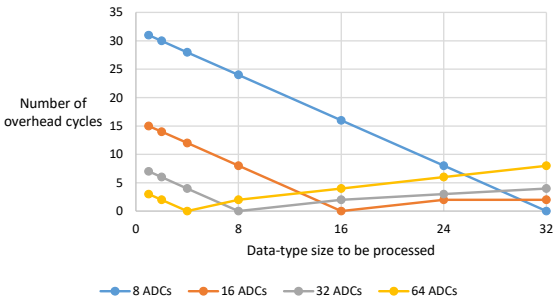


Figure 7.11: The overhead imposed by the first solution for different configurations [220].

- The first solution relies on always shifting enough times to ensure proper result positioning. As the size of the registers is fixed, this can be implemented by having a counter keep track of the number of times the result has been shifted. Once the computation has finished (indicated by the IADD, CP, and CB instructions for the different intermediate registers), the result can be shifted

the remaining number of times. The high-level implementation can be seen in Figure 7.10(a). While this solution offers low implementation complexity, it compromises the performance of the addition scheme heavily for smaller datatypes as the addition scheme now has a fixed latency rather than a latency scaling linearly with the datatype size. The overhead cycles would add directly to the latency of the second pipeline stage, which is already the critical stage due to the read-out part of the stage. Assuming the system is designed for 32-bit, Figure 7.11 shows the number of overhead cycles this solution imposes on each element computation for different numbers of ADCs (corresponding to different temporary register sizes). For each ADC configuration, there is an optimal datatype size for which there is no cycle overhead due to the addition scheme. This corresponds to the datatype size where the computation result fits exactly in the $R4_{temp}$ register and can then be directly copied to the output buffer. For smaller datatype sizes, the overhead comes from the extra shifts required in the $R2_{temp}$ and $R4_{temp}$ registers. For larger datatype sizes, the overhead is due to the required addition between different $R4_{temp}$ registers.

- The second solution does not require any overhead of shifting. In this design, we insert the data in the correct starting position of the intermediate register. The datatype that is being processed is used to select the proper position in which the data should be inserted in the intermediate register. After the number of shifts required for the addition, the data will be in the correct position in the register without any overhead cycles. While this solution does not compromise the addition scheme performance at all, it leads to huge amounts of multiplexing required to select the proper starting position if the CIM-Tile should support any random datatype. To eliminate this unreasonable multiplexing, a constraint can be set on the datatypes that the tile can process. By having the tile only process datatypes, which are multiples of bytes, only four different starting points in the shift register are required to support up to 4-byte datatypes. Figure 7.10(b) illustrates the principle of this solution. The 'adder config' signal sets up the multiplexing to select the proper starting point.

- If the constraint on the datatypes turns out to be a significant issue for future applications, we can consider the third solution. This solution allows for computation on all random datatypes (thus omitting the constraint imposed by solution 2) without imposing large numbers of overhead cycles by using a hybrid design of the first and second solutions. If, for example, computation should be performed on a 10-bit datatype, the multiplexing of solution two can set the starting point in the shift register at 16. Then, after computation, the shifting described in solution one can be used to shift the remaining six positions. This solution allows processing any datatype with a maximum of 7 overhead shifts at the cost of more complex control. As there are, at the time of writing this report, no reasons to believe that the constraint on datatypes is an issue for applications, the second solution is deemed sufficient for the hardware implementation. For every solution, it is required that the 'Addition

Unit' is set up properly to perform computation on the desired datatype. Depending on how the tiles will be deployed, this can be done by either setting up the addition unit before executing a program or allowing for a change of the set-up at run-time with the help of the compiler by introducing a new nano-instruction.

7.4.6. EVALUATION AND DISCUSSION

In this subsection, we evaluate our design against a reference design. First, we introduce the reference design and the target technology. Subsequently, the results regarding the performance and energy of our proposed method will be discussed.

EXPERIMENTAL SETUP

Reference architecture. In the reference design, we assume that a single adder to perform accumulation between shared columns and different bit positions of the multiplier are connected to each ADC. The size of the adder is fixed and must be chosen based on the largest possible value resulting from the MMM - it is specified in Equation 7.3. This means that the value produced by the ADC is merely an intermediate result that must be summed up into the accumulator - remember that only a single bit of the multiplier is multiplied with the multiplicand and each ADC read-out corresponds only to a single bit-position of the multiplicand. Due to the previously stated manner of summation, the intermediate results must be shifted by the correct number of positions (based on the bit-positions of the multiplier and the multiplicand) before entering the adder.

$$\begin{aligned} Adder\ size = &int\ size(multiplier) + int\ size(multiplicand) \\ &+ \log_2(crossbar\ height) \end{aligned} \tag{7.3}$$

Table 7.2: Tile configuration

Crossbar		
Technology	ReRAM (256x256 @1bit)	
Read energy	0.4pj per cell	
Write energy	40pj per cell	
Read/Write latency	100ns	
ADC (8-bit)		
Energy	2pj per sample	
Latency	1ns per sample	
Carry-lookahead Adder		
Size	Energy (per computation)	Latency
8-bit	0.01pj	1ns
16-bit	0.03pj	2.2ns
24-bit	0.08pj	3.2ns
40-bit	0.25pj	5.6ns
72-bit	0.78pj	9.8ns

In order to evaluate our design (against the reference design), we have synthesized both using Cadence Genus with standard cell 90nm UMC library. The latency and energy numbers related to the crossbar (assumed to use ReRAM for now) and (SAR) ADCs are taken from [192] and [193], respectively. These numbers are summarized in Table 7.2, and more detailed information about the ReRAM device can be found in [192]. Subsequently, these numbers serve as input to our in-house tile simulator. Our simulator is capable of executing programs and capturing the behavior of the crossbar and peripheral circuit. As a benchmark, we have chosen the linear-algebra kernel "gemm" from the Polybench/C benchmark suite. In this kernel, first the multiplicands are written into the crossbar and then the actual multiplication is performed. We utilized our in-house developed compiler to translate the gemm benchmark into a sequence of in-memory instructions controlling the crossbar and its peripheral circuits.

EXPERIMENTAL RESULT

Reading data from the crossbar's columns (the read-out phase) is inherently slow mainly because of the shared ADC between multiple columns and can be considered as the bottleneck of the architecture. However, as the size of the adder grows, its latency can be dominant over the latency imposed by ADC. Considering the proposed design in our experiment, since the crossbar has 256 rows, the maximum size of the first two adders is always 8-bit regardless of datatype size. Therefore, there is no extra overhead on the latency of the read-out phase. Rather, in the reference design and according to Equation 7.3, the required size of the adder is much larger. Accordingly, considering a 1 ns latency for the ADC, as the size of the adder increased more than 8-bit, it became the bottleneck of the system and made the read-out phase more costly.

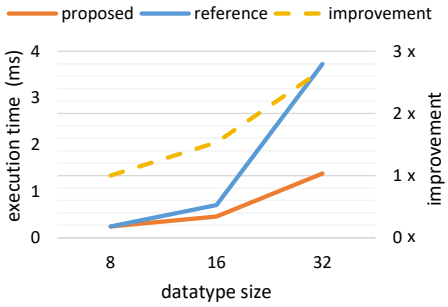


Figure 7.12: Execution time for different integer datatype sizes.

Figure 7.12 depicts the execution time of the kernel (including writing to the crossbar). In this experiment, we assume that the size of the integer numbers matches the number of columns shared by one ADC. According to the figure, the size of the first two adders for the proposed design is always constant (here 8-bit).

Instead, as the datatype size increases, a larger adder has to be employed for the reference design. Considering an 8-bit datatype size in Figure 7.12, a 24-bit adder has to be used for the reference design, which imposes a 3 times bigger latency than an ADC. However, due to the abundance of ADCs, the entire readout phase is not the bottleneck of the system (see Table 7.2 for the crossbar latency). Therefore, there is no performance improvement at this point. Figure 7.13 shows the energy improvement achieved by the proposed design. Although the number of computations is always the same, they are performed with smaller adders, which has a positive impact on the energy consumption of the addition unit. Finally, the impact of the number of ADCs on the execution time and energy of the addition unit using 32-bit datatype size are presented in Figure 7.14. As the number of ADCs increases, the performance is improved. In addition, although more adders are employed (see Figure 7.8), their size is decreased, which leads to less energy consumption.

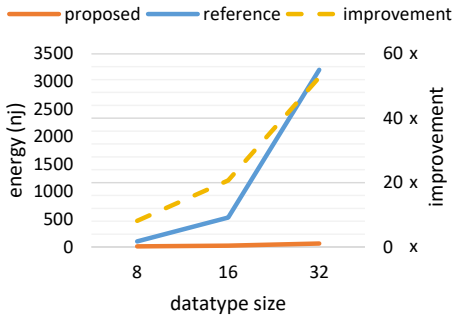


Figure 7.13: Energy consumption of addition unit for different datatype sizes.

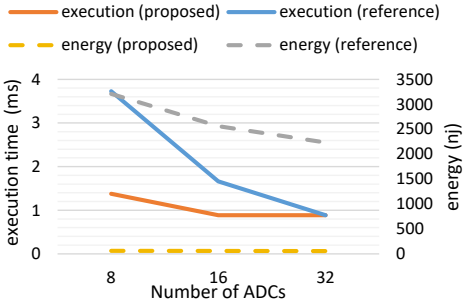


Figure 7.14: Energy consumption of addition unit and execution time of the benchmark for different numbers of ADCs.

7.5. A Novel Signed Data Representation and Computation

In this section, we propose our optimized design to support two's complement representations in a CIM tile. The design uses the same hardware discussed for unsigned representation with some changes in the execution flow.

7.5.1. Motivation

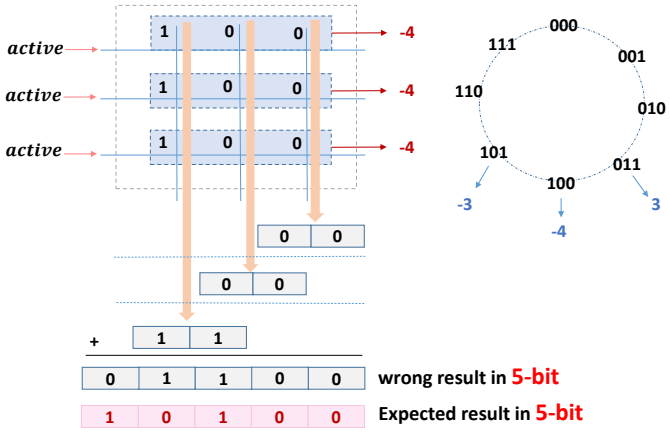


Figure 7.15: Example of sign extension required for two's complement representation when the size of the output is larger than the input data.

Despite other representations, the primitive arithmetic operations in two's complement representation are identical to unsigned numbers. This is particularly helpful when operating on many operands at once. However, considering the crossbar structure, the sign extension required for this representation imposes a big overhead on the system. Figure 7.15 shows an example where the range of data is from -4 up to +3 which can be represented in 3 bits. Assuming each memristor device can hold one bit, 3 devices can represent one number. As we show in this figure, by performing an addition among the numbers programmed to the crossbar, the result is different than expected. Although each number individually can be presented in 3 bits, more bits are required to present the output result since several numbers are summing up. Accordingly, input operands have to be sign-extended to be the same size as the output. In general and in the case of addition operation, the number of bits that have to be considered for sign extension is equal to $log2\ (number\ of\ crossbar\ rows)$. Clearly, these extra bits degrade energy, performance, and area efficiency. Therefore, it is essential to address this challenge and improve the efficiency of signed arithmetic computation in the crossbar.

The sign extension causes an overhead on both array data and input data. In Subsection 7.5.2, we address the challenge of sign extension on array data using a simple arithmetic addition operation as an example. Subsequently, in Subsection

7.5.3, we consider the MMM operation as a more complex operation where the overhead of sign extension exists on both array data and input data. Considering this operation, we extend our solution to address the overhead of sign extension on input data as well.

7.5.2. EFFICIENT SIGNED ADDITION BY INTRODUCING VIRTUAL BIT-LINES

To address the overhead of sign extension on array data, a new method is proposed based on what is named ***virtual bit-lines***. We illustrate our solution in Figure 7.16 using a simple arithmetic addition example where we want to obtain the result of addition between three signed numbers stored in the crossbar.

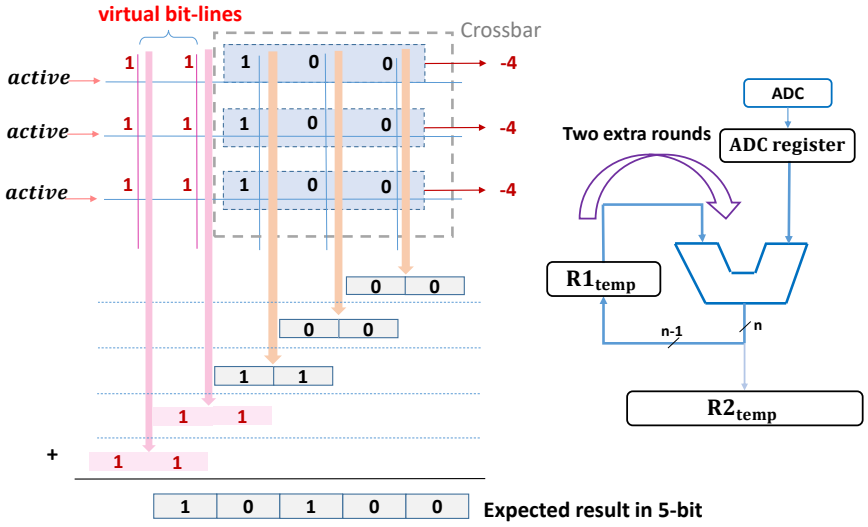


Figure 7.16: Low-cost implementation of signed addition by introducing virtual bit-lines.

Instead of programming more memristor devices to capture the effect of sign extension, we propose a novel method to incorporate the impact of sign extension in the crossbar periphery. As demonstrated in Figure 7.16, the values obtained from the fourth and fifth columns, used to represent sign extension values, are the same as the third column. This is based on the fact that the values of the sign-extended bits are equal to the most significant bit before the sign extension. By recalling how unsigned computation is performed, in each time step, the value of one column is stored in the ADC register. In the third time step, the value of the third column is stored in the *ADC register* and has to be summed up with the intermediate result obtained from previous columns (columns 1 and 2), which are (partially) stored in the $R1_{temp}$ register. Since the values of the virtual bit-lines are equal to the value of the third column, and this already exists in the *ADC*

register, only the loop is required to be performed extra rounds. The loop has to be performed more times equal to the number of virtual bit-lines (in the worst-case $log2(number\ of\ crossbar\ rows)$. By doing this, 1) a fewer number of bit-lines have to be programmed (crossbar programming energy), 2) a fewer number of bit-lines are contributing to the computation (crossbar computation energy), and 3) a fewer number of bit-lines read by ADCs (ADC energy). This leads to a huge improvement in terms of area, performance, and energy efficiency.

7.5.3. Efficient signed multiplication by employing virtual input segments

In the previous subsection, we discussed how to eliminate the overhead of sign extension on array data by only considering its contribution to the result in the periphery. In this subsection, we consider the execution model for a complete MMM operation where the overhead of sign extension exists both on array data and input data. We explain how this affects the number of virtual bit-lines and how we can optimize this operation further.

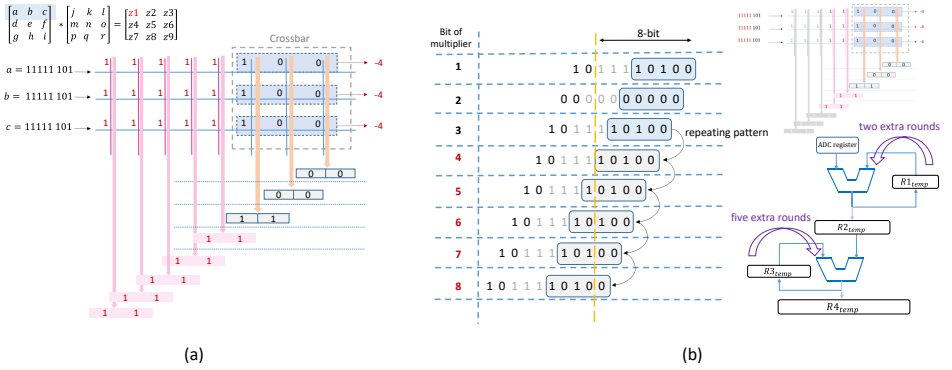


Figure 7.17: Introducing *virtual input segments* besides virtual bit-lines to reduce the overhead of signed MMM (b) applying small modifications on execution flow to support signed MMM while employing the same hardware used for unsigned computation.

Figure 7.17(a) illustrates an example of the MMM operation where the elements of both multiplier and multiplicand are presented in two's complement format bound for three bits. The size of elements for the output matrix (S_{out}), as well as the number of sign extension bits for both multiplicand (E_{mpd}) and multiplier (E_{mpr}), are computed according to Equation 7.4, 7.5, and 7.6. Since the range of elements for the output matrix (8 bits) is larger compared to an addition operation (5 bits), more overhead is imposed due to the sign extension even using the aforementioned technique. However, to reduce this overhead, we propose two optimizations as follows.

$$S_{out} = (\text{multiplicand datatype size}) + (\text{multiplier datatype size}) + \log_2(\text{active rows}) \tag{7.4}$$

$$E_{mpd} = S_{out} - (\text{multiplicand datatype size}) = \text{number of virtual bit-lines} \tag{7.5}$$

$$E_{mpr} = S_{out} - (\text{multiplier datatype size}) = \text{number of virtual input segments} \tag{7.6}$$

❶ Similar to computation over unsigned numbers (Section 7.4), here, input segments, including the sign-extended bits, are given to the crossbar one at a time. In this example, since the output needs to be presented in 8 bits, we have to have 5 bits sign extension for each input operand to bring them into the same size as the output. The bits for the sign extension are indicated with red color in Figure 7.17(a). The result of each segment has to be summed up with the preceding segments to get the final value. Considering this example, Figure 7.17(b) shows the intermediate result obtained by applying each segment (here one bit at a time) to the crossbar. Due to the sign extension, the intermediate results of applying the fourth until the eighth bit of the multiplier are the same as the third bit. Similar to virtual bit-lines, we call these input segments as **virtual input segments**. Therefore, since the result by applying the virtual input segments already computed in the previous rounds and exists in the $R2_{temp}$ register, these segments are not given to the crossbar as input. Looking back on the proposed hardware for unsigned numbers, there are three stages, 1) analog addition, 2) sliding over multiple columns, and 3) sliding over input segments. Therefore, as depicted in Figure 7.17(b), the third stage, which is in charge of sliding over input segments, has to be performed five additional times (equal to the number of virtual input segments). Accordingly, this attains energy and performance improvement due to the fewer activations for the crossbar as well as periphery circuits.

❷ While the first optimization was focused on the input data and the virtual input segments, here we focus on the size of the adders and registers.
According to the aforementioned optimization techniques proposed in Section 7.4, the size of the adder associated with the third stage can be reduced down to the length of an intermediate result obtained for one input segment. However, due to the sign extension of the array data, one can think a larger adder size (e.g., 8 bits) is required since each input segment produces 8 bits as a result. By considering the example in Figure 7.17(b), it can be realized that from the 8-bit value resulting for each bit of input data, the last three (gray color digits) are the same as the fifth bit. This is due to the fact that giving an input bit (in general input segment) to the crossbar is like an add operation (discussed before) where the number of sign extended bits is proportional to only the number of active rows (crossbar rows in

the worst-case) and the extra sign extended bits imposed by the MMM operation results to the bits with the same value. In this example, among five sign extended bits for the array data, **two bits** (or two virtual bit-lines) are due to the number of active rows (or the number of array data elements that are summed up), and the rest are imposed due to the input data in the MMM operation (see Equation 7.5). Hence, in order to get the first five bits of the result related to each bit of the multiplier elements, the second stage of the addition scheme requires performing only two extra rounds like what was mentioned for an add operation (more rounds lead to repeating the MSB). Consequently, we minimize 1) the number of extra rounds required in the second stage and 2) the length of output generated in the second stage, which leads to downsized adders and registers in the third stage.

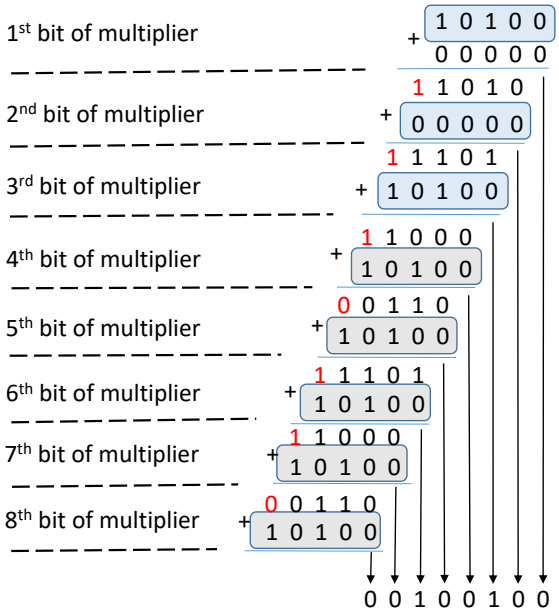


Figure 7.18: Sign extending the output of second stage adder only for one-bit (red digit). This is enough for correct execution and also minimizes the size of adders/registers in the second stage.

After obtaining the intermediate values from the second stage of the addiction scheme (e.g., 5 bits in this example), the third stage has to sum them up. Figure 7.18 depicts how to perform this summation while using the same adder size employed for unsigned operations (in this example 5-bit adder rather than 8-bit adder). As we can see, after each addition, the LSB is directly stored in the $R4_{temp}$ register, and the rest with a one-bit sign extension will be stored in the $R3_{temp}$ register. This one-bit sign extension captures the contribution of sign-extended bits at the output of the second stage when we downsize it from 8 to 5 bits (gray digits in Figure 7.17(b)).

Therefore, using this one-bit sign extension, we ensure the correct functionality of the third stage while reducing the size of the adder from 8-bit to 5-bit.

In order to generalize our discussion, the following equations are provided. The Equations 7.7 and 7.8 calculate the number of extra rounds required for stages 2 and 3. In addition, the size of the adders, as well as the registers for both stages, are shown in Equations 7.9 and 7.10 where $Adder_1$ and $Adder_2$ are the adders employed for stage 2 and 3, respectively. It is worth mentioning that the second term in Equation 7.9 is added when the number of resistance levels is more than two.

$$\begin{aligned}&\text{Second Stage Extra Rounds} = \\ &\log_2(\text{active rows}) \leq \text{number of virtual bitlines}\end{aligned} \tag{7.7}$$

$$\begin{aligned}&\text{Third Stage Extra Rounds} = \\ &E_{mpr} = \text{number of virtual input segments}\end{aligned} \tag{7.8}$$

$$\begin{aligned}&\text{Size}(\text{Adder}_1) = \text{Size}(\text{ADC register}) = \text{Size}\ (\text{R1}_{temp}) \\ &= log2(\text{number of rows}) + log2(\text{resistance levels})\end{aligned} \tag{7.9}$$

$$\begin{aligned}&\text{Size}(\text{Adder}_2) = \text{Size}\ (\text{R2}_{temp}) = \text{Size}\ (\text{R3}_{temp}) = \\ &\text{multiplicand datatype size} + log2(\text{number of rows})\end{aligned} \tag{7.10}$$

7

7.5.4. ASYMMETRIC ADDITION SCHEME

Until now, we assumed that the array data datatype size is aligned or can be fit within the number of columns shared by an ADC. However, the proposed solution is valid even when the assumption does not hold. By increasing the number of ADCs per crossbar, the numbers which are stored in the crossbar might be distributed over several ADCs depending on the application and its datatype size. Figure 7.19 illustrates this scenario by employing a simple example where both input data and array data have 6 bits size and three columns shared by an ADC. As can be seen, while virtual bit-lines have only an implication on the addition unit operating on the most significant bits, the virtual input segments influence both units. Since three rows are considered, two extra rounds– $log2(number\ of\ rows)$– are imposed to the second stage of the addition unit on the left side. Besides, due to the eight segments of virtual input (here each segment is one bit), eight extra rounds have to be performed on the third stage of both units. It must be highlighted that since the virtual bit-lines are not associated with the addition unit on the right side, there is no 1-bit sign extension when storing the intermediate result in the $R3_{temp}$ register, while the addition unit on the left side requires this as explained before (see Figure 7.18.

Considering the aforementioned scenario, when the final result associated with an ADC is achieved and stored in the $R4_{temp}$, they have to be summed up considering

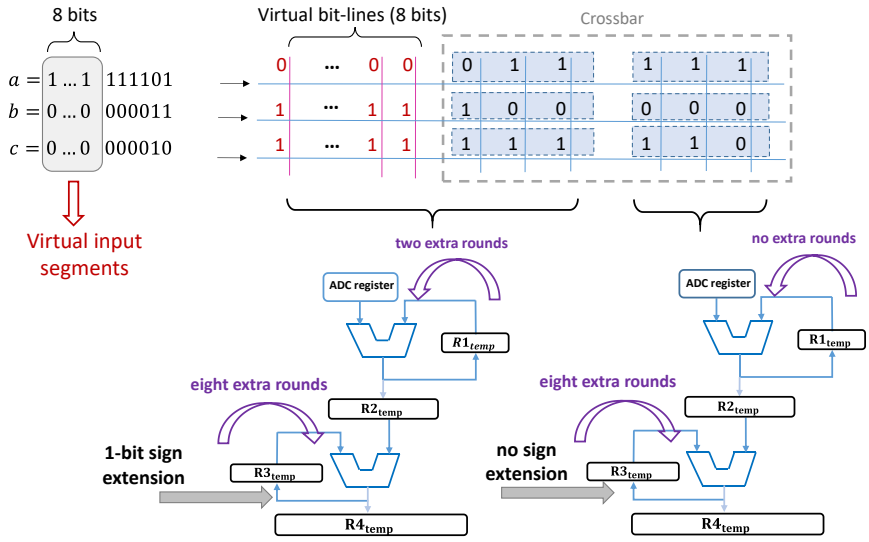


Figure 7.19: Asymmetric procedure for addition scheme when a number is split over several ADCs.

their positions. The same structure explained in section 7.4 is employed here in order to minimize the size of the adder and register in this fourth stage. Figure 7.20 depicts an example where the values of the $R4_{temp}$ registers related to each addition unit are summed up in sequence. In each step, the first three bits of the adder's output are directly stored in the $Rfinal$ register since the values in $R4_{temp}$ registers have three bits positional difference. Consequently, this reduces the size of the intermediate register as well as the adder which in turn improves performance and energy. It is worth mentioning that the processing in this stage can be carried out in parallel with the lower stages. Therefore, performing the computation sequentially in this stage does not induce performance overhead on the system since the previous stages require more time.

7.5.5. Experimental Setup

In this section, we explain how we evaluate energy, performance, and area as the main metrics in our simulation. The design is compared with two baselines on three different benchmarks which cover different scenarios when executing arithmetic operations on the crossbar.

Simulation Platform

The result is based on our SystemC simulator [13, 19], which is parameterized by incorporating power and timing characteristics for the memristor devices, digital, and analog circuits. First, we profile the application to decompose it into sets of MMM operations. In the next step, MMM operations are decomposed again into sets of

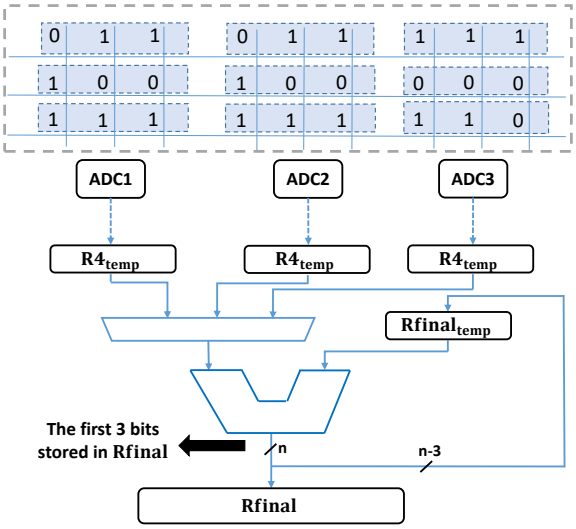


Figure 7.20: Extra stage to sum up the result obtained from addition scheme per ADC. This is illustrated for a simple addition operation, and we only require this hardware when a number is split into multiple ADCs.

in-memory instructions. This is done automatically by our in-memory compiler. The instructions give us programmability and controllability over memristor crossbars. The instructions are then executed by our in-memory simulators, which mimic the behavior of the hardware, including the crossbar and its analog as well as digital peripheries. The simulator reports the simulation time and energy numbers based on the activity factors of different components. The energy and latency numbers for each digital component (e.g., decoder, addition scheme) are obtained by a synthesis tool (Cadence Genus) and then exported into the simulator. The code for our compiler and simulator is available online [229]. All the values for the analog components of the CIM-tile are summarized in Table 7.3. The parameters for ReRAM and PCM technologies are taken from [223] and [82] validated for the actual device. For all the simulations, it is considered each memristor cell can hold one bit (two resistance levels).

Performance:
The simulator executes the in-memory instructions generated by our compiler to control the CIM tile, program the devices, and perform the computations. The instructions are decoded and executed in 1 GHz clock frequency. Assuming one bit stored in each cell, in order to activate all the crossbar rows at the same time, an 8-bit ADC at 32 nm is employed [193]. For all the simulations, each ADC is shared between 8 bit-lines, which directly influences the performance of the system.

Area:
The area of each ReRAM cell in the 1T1R crossbar structure is taken from [178]. The

Table 7.3: Value of parameters used for the analog components.

Component	Parameters	Spec	
clock	frequency	1GHz	
Memristive devices		ReRAM	PCM
	cell levels	2	2
	LRS	5k	20k
	HRS	1M	10M
	read voltage	0.2 V	0.2 V
	write voltage	2 V	1 V
	write current	100 μA	300 μA
	read time	10 ns	10 ns
	write time	100 ns	100 ns
Crossbar	structure	1T1R	
	num. columns	256	
	num. rows	256	
S&H	number	256	
	hold time	9.2 ms	
	latching energy	0.25 pJ	
	latency	0.6 ns	
DAC		read DAC	write DAC
	number	256	256
	power	3.9 μW	3.9 μW
ADC	power	2.6 mW	
	precision	8 bits	
	latency	1.2 GSps	

digital peripheries are synthesized in Cadence Genus targeting standard cell 15 nm Nangate library to obtain the area consumption. The number of required CIM tiles depends on the benchmarks.

Energy:

Since all the information and control signals can be tracked by employing our simulator, a typical **activity factor** and performance number are extracted. These numbers are incorporated to achieve accurate energy consumption for the digital as well as analog components of the tile. The power consumption for ADC was obtained from [193]. Besides, the power consumption of 1-bit DAC is taken from [194]. However, in the simulations where more bits are required, the power consumption is taken from the equations provided in [194]. The power consumption related to the digital periphery is taken from the Cadence Genus report targeting standard cell 15 nm Nangate library.

Benchmarks

When we execute arithmetic matrix-matrix multiplication on the crossbar, there would be two scenarios in the perspective of the signed computation: 1) only one

operand or 2) both operands (input and programming data) have to be expressed as a signed number. The benchmarks are selected to be able to assess the behavior of both scenarios. Considering the first scenario, a neural network is employed to classify the MNIST database. The network has two hidden layers with 80 and 60 neurons, respectively. The network trained, providing around %97 accuracy, and the weights are obtained using MATLAB. The programming weights are presented in 8-bit **signed** fix-point whereas the input data (pixels) is considered as 8-bit **unsigned** number. It is worth mentioning that the intermediate values generated by the hidden layers are considered binary numbers. For the second scenario, we take two kernels from the Polybench/C benchmark suite. The first kernel is linear-algebra "gemm" where the input matrix size is 1000×1200, and the programming data matrix size is 1200×1100. Similarly, the second kernel, "3m", has the problem size of 800×1000 and 1000×900 for the input and programming data, respectively. The datatype size for the aforementioned kernels is 8-bit, and despite the first benchmark, both input and programming data are presented as **signed** numbers. Since we only focus on the crossbar and its periphery circuit in this paper and not a full system simulation, considering more benchmarks will not add more insight to our study, and a similar pattern will be repeated.

BASELINES

The proposed design is compared against two baselines to quantify its efficiency in performing signed computations in terms of energy, performance, and area. The baselines are as follows:

- Baseline1:
  In this baseline, both the inputs [67] and the programming data [65] are presented as a standard two's complement representation where the sign extension is required for both multiplier and multiplicand in the case of MMM operation. Furthermore, in order to have a fair comparison, we employ the same efficient structure proposed for the *addition unit* (see Section 7.4) rather than conventional *Shift-and-add* structure. Therefore, the result will completely focus on the way signed computation is performed. This *addition unit* with the proposed structure is solely evaluated in previous section against conventional *Shift-and-add* structure.

- Baseline2:
  In this approach, the positive and negative programming data are mapped into two different crossbars where the data on each crossbar is treated as unsigned value [69, 110]. Furthermore, in order to support signed input, the current is provided in two directions to represent negative and positive numbers (see Figure 7.3(a)). For all the simulations regarding this baseline, we consider 2-bit DACs to provide three voltage levels corresponding to logical -1, 0, and 1 values. Finally, the same *addition unit* considered for the first baseline is also employed here.

7.5.6. Results

In this section, we evaluate the proposed design in terms of execution time, area, and computation, as well as programming energy. In the following, the result regarding each of the aforementioned aspects of the design is discussed to provide a good insight into its pros and cons.

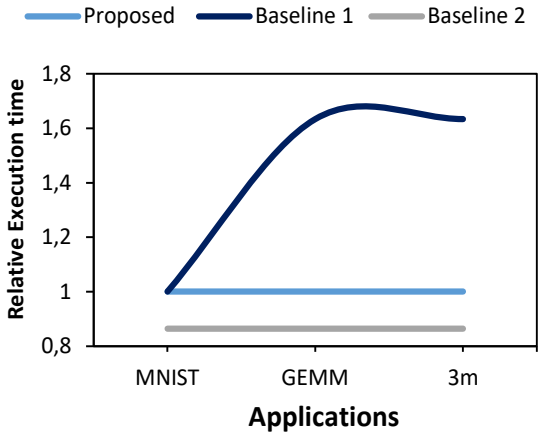


Figure 7.21: Relative execution time of the baselines normalized to the proposed design. No distinction between the two memristor technologies (According to Table 7.3, the two memristor technologies have similar latency numbers).

Execution time

Figure 7.21 depicts the relative execution time for the two baselines normalized with the proposed scheme. As can be seen in this figure, due to the extra rounds of addition required in the proposed scheme (see Figure 7.17), a small overhead is imposed on the execution time compared to Baseline 2. According to the description of Baseline 2, since the drivers have to generate three voltage levels, they may have more latency than the drivers employed for the proposed design as well as Baseline 1. This may have an adverse impact on the performance of Baseline 2 and reduce the gap between this scheme and the proposed design. Considering MNIST, Baseline 1 and the proposed scheme attain a similar performance since either eight extra crossbar columns (Baseline 1) or eight extra rounds in the *addition unit* (proposed design) has to be performed. This is due to the sign extension. However, if the digital clock frequency gets bigger than the ADC conversion rate, the overhead of extra rounds in the *addition unit* will be reduced. Regarding GEMM and 3m, since the inputs are also signed, the execution time for the Baseline 1 is increased due to the costly sign extension of input data. It is worth mentioning that the overhead of data communication to provide input data to the crossbar is considered for all the designs. Based on the architecture proposed in [14], in order to provide a

clear separation of tile from outside and minimize the number of synchronizations, a buffer called *Row Data Buffer* with the width equal to the maximum supported datatype size, and the heights equal to the number of crossbar rows is placed next to the crossbar. This buffer provides input data for the crossbar. Filling each row of this buffer with the elements of the multiplier matrix takes one clock cycle.

As mentioned before, when we execute arithmetic matrix-matrix multiplication on the crossbar, there would be two scenarios in the perspective of the signed computation: 1) only one operand or 2) both operands (input and programming data) have to be expressed as a signed number. As can be seen in Figure 7.21, for the scenario where the inputs are unsigned (MNIST), the proposed design cannot outperform Baseline 2 in terms of performance. Baseline 2 maps positive and negative weights into two different crossbars. Hence, the required computations are performed in parallel in two crossbars and their peripheries. However, this computation is performed sequentially in the periphery of the proposed design. Although this could significantly improve energy and area consumption (we discuss it in the following), it brings marginal performance loss. It should be noted that this only happens in the scenario where the inputs are unsigned numbers.

AREA

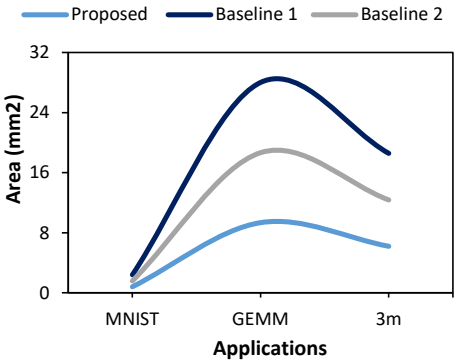


Figure 7.22: Area comparison of the Baselines with the proposed design for ReRAM crossbars taking into account the digital and analog components.

Figure 7.22 depicts the area consumed by the three designs. Regarding the first baseline, since we stored the sign-extension bits in the crossbar, more crossbars have to be employed, and in consequence, more area is consumed. In the second baseline, we do not need this costly sign extension, but since positive and negative elements in a matrix have to be assigned to different crossbars, it consumes more area than the proposed design. Finally, the actual area number for each design depends on the problem size of the benchmarks. However, the ratio between these designs remains constant for different benchmarks. It should be noted that the area of digital peripheries is also considered in this figure for both the proposed design

as well as the baselines. These numbers are based on the 15 nm Nangate library.

ENERGY

Computation energy:
Figure 7.23(a) demonstrates the computation energy for the aforementioned benchmarks considering the baselines and the proposed design. The results are presented for PCM and ReRAM technologies without considering the programming energy.

MNIST: For the MNIST benchmark, the experiment is performed for 10k input images, and as can be seen, more than 3× energy improvement is obtained compared to the baselines. In the following, we analyze the results obtained for each baseline.

- Regarding the first baseline and considering the 8-bit datatype size, the programming weights should be sign-extended to 24 bits (only the programming weights are signed, not inputs). This is because the inputs are 8 bits unsigned, and the total number of elements on each column of the crossbar is 256. Hence, the output size would be $24 = log2(256) + 8$. According to the output size, the weights should be sign extended to 24 bits. The implication is the necessity of more crossbar arrays together with their peripheries to encompass the weights. This leads to more energy consumption, as shown in Figure 7.23(a).

- Considering the second baseline, the number of crossbar arrays is doubled due to allocating positive and negative weights into different crossbars. Although the inputs for this application are not signed, the platform considered for this baseline has a higher precision of DACs and ADCs to support negative inputs, which imposes energy overhead on the system. It should be taken into account that the ADCs need to be sensitive enough in this baseline to detect finer current/voltage values in order to keep the accuracy as high as required. In conclusion, for this baseline, both the crossbar and peripheries contribute more to energy consumption than the proposed design.

Furthermore, we perform the analysis for two memristor technologies, ReRAM and PCM. According to Table 7.3, PCM has higher resistance values for both low (LRS) and high (HRS) states. Therefore, having the same 'read voltage', the energy consumed by this device is less than ReRAM during the computation. We can observe this in Figure 7.23(a), where we report the energy consumption for these two technologies.

GEMM and 3m: The problem sizes for these two kernels are larger than MNIST. In addition, the inputs and programming weights are 8-bit signed numbers. Therefore, considering the first baseline, both the inputs and programming weights should be sign-extended to 24 bits. Hence, not only more crossbars are required, but more input segments should be given to the crossbars as well. This leads to around 8× more energy consumption compared to the proposed design. Despite the first baseline, the approach used in the second baseline can deal with signed inputs with

less overhead. Comparing the proposed design with the second baseline and similar to MNIST, the improvement is mainly obtained from employing fewer crossbars as well as lower DACs and ADCs precision.

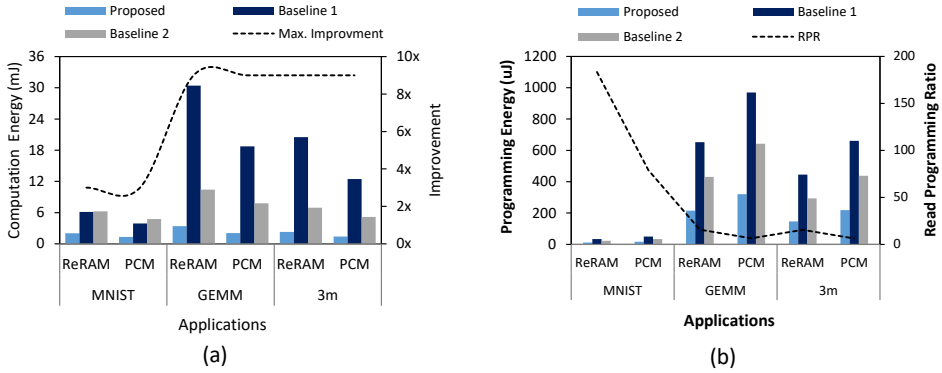


Figure 7.23: Energy consumption for (a) computation and (b) programming phase achieved for the two baselines and the proposed design. The energy comparison among different designs is performed for PCM and ReRAM technology.

7

Programming energy:
Besides the high energy consumption of programming the memristive devices, the endurance of these devices is a critical feature [230], which persuades the users to program them as minimum as possible. Therefore, usually, it is simply considered that the crossbar is programmed once before the execution of the application without (or with minimum) reprogramming during the execution. However, it is necessary for the designer to have insight into the overhead of crossbar programming compared to the actual computation phase. Hence, Figure 7.23(b) provides the programming energy as well as the ratio of energy consumption for the computation over the programming phase.

Considering MNIST and after the classification of 10K input images, the result shows that the overhead of programming is getting negligible compared to the computation phase. However, this overhead is a bit higher for PCM devices due to the higher programming current. In addition, since the proposed design requires fewer crossbars, the energy overhead imposed by device programming is lessened compared to the baselines. This might also have positive implications on performance and aging which has not been considered in this paper. Since GEMM and 3m kernels have a larger problem size (require more crossbars) and relatively less computation, the overhead of programming is increased more. It should be noted that, similar to MNIST, these two kernels can also be executed multiple times for different input matrices. Therefore, based on the system requirements and the

application data flow, the designer should evaluate when and for how many times the devices can be reprogrammed.

7.5.7. LIMITATIONS AND CHALLENGES

The proposed design achieves 8× and 3× energy and area efficiency. However, this comes with two main drawbacks. In the following, we provide a short explanation for each of them.

1. Performance limitation:
   The proposed design has marginally lower performance compared to baseline 2 where positive and negative weights are mapped into two different crossbars. This is because the required computations are broken into two parts and performed in parallel. However, more sequential operations are performed in the peripheries of the proposed design. This performance loss also depends on the clock frequency of digital circuits. If the clock frequency is high enough, the latency of the crossbar can hide the latency of digital peripheries.

2. Complexity of the controller:
   The proposed design can flexibly support different datatype sizes. This comes at the cost of a more complex controller. There are two approaches to control this design 1) implementing a state machine or 2) using instructions. The second approach relies more on the compiler and reduces the complexity of the hardware. However, more complex hardware is still required compared to the baselines, with no flexibility in supporting different data sizes.

7.6. CONCLUSION

In this chapter, we focused on the digital periphery of the crossbar which is responsible to get the crossbar output and generate the final result for arithmetic multiplication and addition operations. First, we proposed a new organization for a memristor-based crossbar periphery to support unsigned integer matrix-matrix multiplication at the tile level. Besides the analog additions in the crossbar, digital additions are performed in a way that minimum adder sizes are required. Second, we proposed a new scheme based on two's complement representation to perform signed arithmetic matrix multiplication tailored for memristor-based crossbar array structure. The key insight of this novel scheme is that with the help of an innovative digital periphery design, the required sign extension can be avoided. The design provides flexibility to perform computation over different datatype sizes as well as switching between sign and unsigned computation at run time.

8

CONCLUSION

This chapter first gives the summary of the thesis. Then, we discuss the future directions.

8.1. SUMMARY

- **Chapter 1: Introduction** This chapter first described the traditional computing systems based on Von-Neuman architecture. We explained the memory hierarchy as well as the limitations and the walls we are facing both from the technology and architecture levels. We presented the concept of memristor-based computation-in-memory, which is considered a promising candidate for future computing platforms. Afterward, we discussed the limitations and challenges of the CIM concept arising at different levels. Considering CIM, we explained there would be two different approaches at two ends of a spectrum where we either have a generic or application-specific CIM-based design. Then, we formulated the general research topic of this thesis to investigate both directions. After that, we summarized the thesis contributions, followed by the thesis organization.

- **Chapter 2: Background and Classification** In this chapter, we first explained the fundamental of memristor devices and three main technologies (ReRAM, PCM, and STT-MRAM). We provided an overview of different memory technology to understand better where the memristor technologies stand with respect to conventional memory technologies. Then, we reviewed on main memory structures proposed for memristor devices. Afterward, we proposed our generic illustration of CIM-Tile, which comprises a memory crossbar and peripheries where the existing works are classified based on that. Thereafter, we explained different system design approaches when it comes to CIM. We observed two general trends: 1) Application-Specific design, where we use CIM for an application under certain assumptions and limitations, and 2) A more generic approach where CIM can be employed for a wider range of applications. Based on this discussion, we explained and elaborated on some existing works.

- **Chapter 3: Application-Specific Design: Neural Networks** This chapter and the following chapters focused on application-specific designs for memristor-based

CIM. This enhances our understanding of CIM, and the behavior of different applications, which ultimately helps us to move further toward our generic path for CIM. We focused on Binary Neural Networks (BNNs) since this is a promising application for embedded systems with hard constraints on energy and computing power. In this chapter, we proposed a novel in-memory memristor-based design that substantially improves both the latency and energy efficiency of BNN networks on CIM. The proposed XNOR-based BNN design replaces the ADC and digital post-processing functionality with a SA with adjusted reference(s) while maximizing parallelization and resource utilization in the design using a novel mapping of weights and activation values in the crossbar and its input buffer. The impact of SA references on the accuracy has been evaluated at the crossbar and network levels. The design was also evaluated in terms of energy and latency for different networks and datasets. This work is able to improve energy and latency up to 5× and 100× compared to the baselines with marginal accuracy loss.

- **Chapter 4: Application-Specific Design: Graph Processing** We studied the potential of CIM for graph processing. Graph processing is employed in a wide range of areas. Poor locality or random access pattern to the memory, in addition to a simple and small amount of computation over the accessed data, make them a good candidate to use CIM. However, the non-optimal use of high sparsity of such datasets leads to a waste of resources as the computation is also performed on zero's operands which do not contribute to the final result. In this chapter, we focused on graph processing applications and designed a novel accelerator, SparseMEM, targeting sparse datasets by leveraging the computing-in-memory (CIM) concept. The proposed solution stores the graph information in a compressed hierarchical format inside the memory and adjusts the workflow based on this new mapping. This vastly improves resource utilization, leading to higher energy and permanence efficiency. The experimental results confirmed the potential of CIM for this application domain.

- **Chapter 5: Application-Specific Design: Bioinformatics** The third and last application domain that we studied is Bioinformatics. The knowledge of bioinformatics is used in a wide range of applications. Food industries are leveraging this knowledge for food profiling which is an essential step in any food monitoring system. Food profilers work on massive data structures and incur considerable data movement for a real-time monitoring system. We translated the problem of food profiling into hyperdimensional computing representation, which makes it easy to be implemented using CIM. Based on that, we proposed our accelerator. We synthesized the required hardware for our accelerator using UMC's 65nm library by considering an accurate PCM model. Our evaluations demonstrate that our CIM-based implementation achieves a (1) throughput improvement of 192× and 724× and (2) memory reduction of 36× and 33× compared to two state-of-the-art profilers (Kraken2 and MetaCache).

- **Chapter 6: Tile Architecture and Simulator** This chapter focused on how to move toward a generic CIM-Tile. We discussed our proposed (micro) instructions to bring programmability into perspective. We provided an overview of the

CIM-Tile and its components associated with different instructions. The detailed implementation of the tile and its controller was described. Besides, to improve the performance, we investigated the possible ways to pipeline the CIM-Tile. Afterward, we briefly explained the compiler, which generates the instruction from higher-level kernels, as well as the simulator. We verified the functionality of the design on FPGA. Finally, the design is evaluated regarding energy and area for an ASIC implementation.

- **Chapter 7: Efficient Digital Periphery Design for CIM-Tile** To proceed with our generic approach toward CIM-Tile, we focused on the digital periphery of the crossbar, 'Addition Unit', in this chapter. This unit performs additional processing to get the crossbar output and generate the final result for arithmetic multiplication and addition operations. The first half of the chapter focused on supporting unsigned computation. The developed structure not only provides a high level of flexibility for us, but also performs the operations in a very optimized manner. This structure forms the base for the second half of the chapter, where we talked about signed computation. we proposed a new scheme based on two's complement representation to perform signed arithmetic matrix multiplication tailored for memristor-based crossbar array structure. The operand mapping is performed without the need for sign extension; hence, reducing the required memory size. This unit is controlled with our (micro) ISA discussed in Chapter 6. The flexibility of the design allows applications to dynamically switch between signed and unsigned computation as well as different datatype sizes without changing the hardware or mapping of data. This is aligned with our approach for a generic CIM-Tile design.

8.2. OUTLOOK

Until now, the main challenges that hinder CIM from being utilized for a large-scale platform are concerned with technology and device abstraction levels. However, scientists from higher abstraction levels can still continue their research with more focus on the two following points:

- **System Design**: During this thesis, we have studied several applications and have designed an accelerator for each. Each accelerator is designed based on several assumptions. Any deviation from them or any changes to the applications may need a new design, which can question the applicability of this approach. A more applicable approach is moving toward more generic and flexible designs by creating some room for changes. As an example, a flexible and generic design can focus on neural network applications while still being able to deal with different network structures and different dataflows. The approach taken during Chapters 6 and 7 to make a programmable design seems promising. This helps us to deal with different technology constraints, applications, and assumptions, while not imposing a big overhead on the system. However, what was presented in this thesis was only one step toward a complete system. The following might be a concern for future research direction.

1) Multi CIM-Tile: To realize CIM for real applications, the system should consist of many CIM-Tiles. However, it should be studied how the data flows from tiles; what would be the proper network structure, and how the system is controlled at this level.
2) Integration to general-purpose processing: The functionality of the CIM-Tile is limited. Considering real-world applications, some parts of them may or cannot be efficient to be executed on CIM-Tiles. Besides, based on our study of different applications, there might be some 'auxiliary' operations within the program that can be offloaded on CIM-Tile that are generally not supported by CIM-Tiles (e.g., Sigmoid function in NN, Min, and Max in Graph, Shift in Bioinformatics). Hence, a researcher should find a solution to address this problem. One possible direction is to consider a general-purpose CPU to support these extra functionalities that are not supported by CIM-Tiles, or another alternative is tightly coupling FPGA with CIM-Tiles. In any case, researchers should define how these generic compute units are integrated and how they interface with CIM-Tiles. The researchers should determine to what extent we can rely on these generic compute units since frequent execution of operations on them may increase the cost of data movement and diminish the benefit of CIM.

- **Simulation platform**: The current simulation platform is able to mimic the behavior of a single CIM-Tile. This supports many parameters and tends to model everything accurately. This is very beneficial, especially when the device technology is not mature, which helps us to bring its impact up to the micro-architecture level and perform design space exploration. However, scaling up this simulator and running a real-world application may not be applicable due to the high simulation time. Therefore, researchers may think of two levels of simulation platforms. A low-level simulation platform where we accurately model a single CIM-Tile regarding energy, latency, accuracy, or any other concerned metrics. A high-level simulation platform where we model the entire system and execute the entire application while a detail of a single CIM-Tile is abstracted away by receiving the input from a low-level simulator. Moreover, more development should be performed on the compiler. A compiler should extract parts of the application that can be accelerated by CIM-Tiles and map them efficiently into the platform.

BIBLIOGRAPHY

[1] S. Srikanth, L. Subramanian, S. Subramoney, T. M. Conte, and H. Wang. "Tackling memory access latency through dram row management". In: *Proceedings of the International Symposium on Memory Systems*. 2018, pp. 137–147.

[2] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. "Memory Power Management via Dynamic Voltage/Frequency Scaling". In: *Proceedings of the 8th ACM International Conference on Autonomic Computing*. ICAC '11. Karlsruhe, Germany: ACM, 2011, pp. 31–40. ISBN: 978-1-4503-0607-2. DOI: 10.1145/1998582.1998590. URL: http://doi.acm.org/10.1145/1998582.1998590.

[3] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz. "Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake". In: *IEEE Micro* 37.2 (Mar. 2017), pp. 52–62. DOI: 10.1109/MM.2017.38.

[4] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. "Dark silicon and the end of multicore scaling". In: *Proceedings of the 38th annual international symposium on Computer architecture*. 2011, pp. 365–376.

[5] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. "CPU DB: Recording Microprocessor History". In: *Commun. ACM* 55.4 (Apr. 2012), pp. 55–63. ISSN: 0001-0782. DOI: 10.1145/2133806.2133822. URL: https://doi.org/10.1145/2133806.2133822.

[6] S. Salahuddin, K. Ni, and S. Datta. "The era of hyper-scaling in electronics". In: *Nature Electronics* 1.8 (2018), pp. 442–450.

[7] F. Oboril, R. Bishnoi, M. Ebrahimi, and M. B. Tahoori. "Evaluation of hybrid memory technologies using SOT-MRAM for on-chip cache hierarchy". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.3 (2015), pp. 367–380.

[8] S. Yu and P.-Y. Chen. "Emerging memory technologies: Recent trends and prospects". In: *IEEE Solid-State Circuits Magazine* 8.2 (2016), pp. 43–56.

[9] X. Yang, B. Taylor, A. Wu, Y. Chen, and L. O. Chua. "Research progress on memristor: From synapses to computing systems". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 69.5 (2022), pp. 1845–1857.

[10] C. Bengel, A. Siemon, F. Cüppers, S. Hoffmann-Eifert, A. Hardtdegen, M. von Witzleben, L. Hellmich, R. Waser, and S. Menzel. "Variability-aware modeling of filamentary oxide-based bipolar resistive switching cells using SPICE level compact models". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 67.12 (2020), pp. 4618–4630.

[11] M. Fieback. "Testing RRAM and Computation-in-Memory Devices: Defects, Fault Models, and Test Solutions". In: (2022).

[12] T. Hirtzlin, M. Bocquet, B. Penkovsky, J.-O. Klein, E. Nowak, E. Vianello, J.-M. Portal, and D. Querlioz. "Digital biologically plausible implementation of binarized neural networks with differential hafnium oxide resistive memory arrays". In: *Frontiers in neuroscience* 13 (2020), p. 1383.

[13] M. Zahedi, T. Shahroodi, G. Custers, A. Singh, S. Wong, and S. Hamdioui. "System Design for Computation-in-Memory: From Primitive to Complex Functions". In: *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*. 2022, pp. 1–6. DOI: 10.1109/VLSI-SoC54400.2022.9939571.

[14] M. Zahedi, R. van Duijnen, S. Wong, and S. Hamdioui. "Tile Architecture and Hardware Implementation for Computation-in-Memory". In: *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2021, pp. 108–113. DOI: 10.1109/ISVLSI51109.2021.00030.

[15] S. Diware, A. Gebregiorgis, R. V. Joshi, S. Hamdioui, and R. Bishnoi. "Unbalanced bit-slicing scheme for accurate memristor-based neural network architecture". In: *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE. 2021, pp. 1–4.

[16] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, *et al.* "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 715–731.

[17] G. Krishnan, S. K. Mandal, C. Chakrabarti, J.-S. Seo, U. Y. Ogras, and Y. Cao. "Impact of on-chip interconnect on in-memory acceleration of deep neural networks". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 18.2 (2021), pp. 1–22.

[18] L. Xia, B. Li, T. Tang, P. Gu, P.-Y. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, and H. Yang. "MNSIM: Simulation platform for memristor-based neuromorphic computing system". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.5 (2017), pp. 1009–1022.

[19] *MNEMOSENE project*. http://www.mnemosene.eu. Accessed: 2010-09-30.

[20] A. Drebes, L. Chelini, O. Zinenko, A. Cohen, H. Corporaal, T. Grosser, K. Vadivel, and N. Vasilache. "TC-CIM: Empowering Tensor Comprehensions for Computing-In-Memory". In: *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*. 2020.

[21] 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: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 18.3 (2022), pp. 1–24.

[22] M. Zahedi, G. Custers, T. Shahroodi, G. Gaydadjiev, S. Wong, and S. Hamdioui. “SparseMEM: Energy-efficient Design for In-memory Sparse-based Graph Processing”. In: *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10137303.

[23] T. Shahroodi, M. Zahedi, C. Firtina, M. Alser, S. Wong, O. Mutlu, and S. Hamdioui. “Demeter: A Fast and Energy-Efficient Food Profiler Using Hyperdimensional Computing in Memory”. In: *IEEE Access* 10 (2022), pp. 82493–82510. DOI: 10.1109/ACCESS.2022.3195878.

[24] M. Zahedi, T. Shahroodi, C. Escuin, G. Gaydadjiev, S. Wong, and S. Hamdioui. “BCIM: Efficient Implementation of Binary Neural Network Based on Computation in Memory”. In: *submitted to IEEE Transactions on Emerging Topics in Computing (TETC)* 11 (2023), pp. 33964–33978. DOI: 10.1109/xxxxx.

[25] *IBS. As Chip Design Costs Skyrocket, 3 nm Process Node Is in Jeopardy. 2020. Available online (accessed on 16 July 2023):* URL: www.extremetech.com/computing/272096-3nm-process-node.

[26] M. Zahedi, M. Mayahinia, M. Abu Lebdeh, S. Wong, and S. Hamdioui. “Efficient Organization of Digital Periphery to Support Integer Datatype for Memristor-Based CIM”. In: *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2020, pp. 216–221. DOI: 10.1109/ISVLSI49217.2020.00047.

[27] M. Zahedi, T. Shahroodi, S. Wong, and S. Hamdioui. “Efficient Signed Arithmetic Multiplication on Memristor-Based Crossbar”. In: *IEEE Access* 11 (2023), pp. 33964–33978. DOI: 10.1109/ACCESS.2023.3263259.

[28] J. J. Yang, N. P. Kobayashi, J. P. Strachan, M.-X. Zhang, D. A. Ohlberg, M. D. Pickett, Z. Li, G. Medeiros-Ribeiro, and R. S. Williams. “Dopant control by atomic layer deposition in oxide films for memristive switches”. In: *Chemistry of Materials* 23.2 (2011), pp. 123–125.

[29] S. Brivio, G. Tallarida, D. Perego, S. Franz, D. Deleruyelle, C. Muller, and S. Spiga. “Low-power resistive switching in Au/NiO/Au nanowire arrays”. In: *Applied Physics Letters* 101.22 (2012), p. 223510.

[30] K. Oka, T. Yanagida, K. Nagashima, M. Kanai, T. Kawai, J.-S. Kim, and B. H. Park. “Spatial nonuniformity in resistive-switching memory effects of NiO”. In: *Journal of the American Chemical Society* 133.32 (2011), pp. 12482–12485.

[31] J. H. Nickel, J. P. Strachan, M. D. Pickett, C. T. Schamp, J. J. Yang, J. A. Graham, and R. S. Williams. “Memristor structures for high scalability: Non-linear and symmetric devices utilizing fabrication friendly materials and processes”. In: *Microelectronic engineering* 103 (2013), pp. 66–69.

[32] Y. H. Do, J. S. Kwak, Y. C. Bae, J. H. Lee, Y. Kim, H. Im, and J. P. Hong. “TiN electrode-induced bipolar resistive switching of TiO2 thin films”. In: *Current Applied Physics* 10.1 (2010), e71–e74.

[33] B. Govoreanu, G. S. Kar, Y. Chen, V. Paraschiv, S. Kubicek, A. Fantini, I. Radu, L. Goux, S. Clima, R. Degraeve, *et al.* “10× 10nm 2 Hf/HfO x crossbar resistive RAM with excellent performance, reliability and low-energy operation”. In: *2011 International Electron Devices Meeting*. IEEE. 2011, pp. 31–6.

[34] H. Kim, P. C. McIntyre, C. On Chui, K. C. Saraswat, and S. Stemmer. “Engineering chemically abrupt high-k metal oxide/ silicon interfaces using an oxygen-gettering metal overlayer”. In: *Journal of applied physics* 96.6 (2004), pp. 3467–3472.

[35] Z. Wei, Y. Kanzawa, K. Arita, Y. Katoh, K. Kawai, S. Muraoka, S. Mitani, S. Fujii, K. Katayama, M. Iijima, *et al.* “Highly reliable TaOx ReRAM and direct evidence of redox reaction mechanism”. In: *2008 IEEE International Electron Devices Meeting*. IEEE. 2008, pp. 1–4.

[36] C. Chen, C. Song, J. Yang, F. Zeng, and F. Pan. “Oxygen migration induced resistive switching effect and its thermal stability in W/TaO x/Pt structure”. In: *Applied Physics Letters* 100.25 (2012), p. 253509.

[37] N. M. Muhammad, N. Duraisamy, K. Rahman, H. W. Dang, J. Jo, and K. H. Choi. “Fabrication of printed memory device having zinc-oxide active nano-layer and investigation of resistive switching”. In: *Current Applied Physics* 13.1 (2013), pp. 90–96.

[38] F.-C. Chiu. “Resistance switching characteristics in ZnO-based nonvolatile memory devices”. In: *Advances in Materials Science and Engineering* 2013 (2013).

[39] H. Kim, M. P. Sah, and S. P. Adhikari. “Pinched hysteresis loops is the fingerprint of memristive devices”. In: *arXiv preprint arXiv:1202.2437* (2012).

[40] D. Apalkov, B. Dieny, and J. M. Slaughter. “Magnetoresistive random access memory”. In: *Proceedings of the IEEE* 104.10 (2016), pp. 1796–1830.

[41] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, *et al.* “Spin-transfer torque magnetic random access memory (STT-MRAM)”. In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 9.2 (2013), pp. 1–35.

[42] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, *et al.* “A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM”. In: *IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest.* IEEE. 2005, pp. 459–462.

[43] T. Kawahara, K. Ito, R. Takemura, and H. Ohno. “Spin-transfer torque RAM technology: Review and prospect”. In: *Microelectronics Reliability* 52.4 (2012), pp. 613–627.

[44] J. S. Kwak, Y. H. Do, Y. C. Bae, H. Im, and J. P. Hong. “Reproducible unipolar resistive switching behaviors in the metal-deficient CoOx thin film”. In: *Thin Solid Films* 518.22 (2010), pp. 6437–6440.

[45] K.-L. Lin, T.-H. Hou, J. Shieh, J.-H. Lin, C.-T. Chou, and Y.-J. Lee. “Electrode dependence of filament formation in HfO2 resistive-switching memory”. In: *Journal of Applied Physics* 109.8 (2011), p. 084104.

[46] H. Lv, M. Yin, Y. Song, X. Fu, L. Tang, P. Zhou, C. Zhao, T. Tang, B. Chen, and Y. Lin. “Forming Process Investigation of Cu_xO Memory Films”. In: *IEEE electron device letters* 29.1 (2007), pp. 47–49.

[47] M. Yin, P. Zhou, H. Lv, T. Tang, B. Chen, Y. Lin, A. Bao, and M. Chi. “Enhancement of endurance for Cu_xO based RRAM cell”. In: *2008 9th International Conference on Solid-State and Integrated-Circuit Technology*. IEEE. 2008, pp. 917–920.

[48] I. Baek, M. Lee, S. Seo, M. Lee, D. Seo, D.-S. Suh, J. Park, S. Park, H. Kim, I. Yoo, *et al.* “Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses”. In: *IEDM Technical Digest. IEEE International Electron Devices Meeting, 2004.* IEEE. 2004, pp. 587–590.

[49] N. Yamada, E. Ohno, K. Nishiuchi, N. Akahira, and M. Takao. “Rapid-phase transitions of GeTe-Sb2Te3 pseudobinary amorphous thin films for an optical disk memory”. In: *Journal of Applied Physics* 69.5 (1991), pp. 2849–2856.

[50] P. Zuliani, E. Varesi, E. Palumbo, M. Borghi, I. Tortorelli, D. Erbetta, G. Dalla Libera, N. Pessina, A. Gandolfo, C. Prelini, *et al.* “Overcoming Temperature Limitations in Phase Change Memories With Optimized $Ge_xSb_yTe_z$”. In: *IEEE transactions on electron devices* 60.12 (2013), pp. 4020–4026.

[51] C. Ahn, B. Lee, R. G. Jeyasingh, M. Asheghi, G. Hurkx, K. E. Goodson, and H.-S. Philip Wong. “Crystallization properties and their drift dependence in phase-change memory studied with a micro-thermal stage”. In: *Journal of Applied Physics* 110.11 (2011), p. 114520.

[52] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou. “Memory devices and applications for in-memory computing”. In: *Nature nanotechnology* 15.7 (2020), pp. 529–544.

[53] L. Wu, S. Rao, M. Taouil, E. J. Marinissen, G. S. Kar, and S. Hamdioui. “Characterization and fault modeling of intermediate state defects in STT-MRAM”. In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2021, pp. 1717–1722.

[54] A. Gebregiorgis, A. Singh, S. Diware, R. Bishnoi, and S. Hamdioui. “Dealing with non-idealities in memristor based computation-in-memory designs”. In: *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE. 2022, pp. 1–6.

[55] P.-Y. Chen and S. Yu. “Technological benchmark of analog synaptic devices for neuroinspired architectures”. In: *IEEE Design & Test* 36.3 (2018), pp. 31–38.

[56] A. Singh, R. Bishnoi, R. V. Joshi, and S. Hamdioui. “Referencing-in-array scheme for RRAM-based CIM architecture”. In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2022, pp. 1413–1418.

[57] S. Kvatinsky *et al.* “MAGIC—Memristor-aided logic”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 61.11 (2014), pp. 895–899.

[58] G. Snider. “Computing with hysteretic resistor crossbars”. In: *Applied Physics A* 80 (2005), pp. 1165–1172.

[59] M. A. Lebdeh, U. Reinsalu, H. A. Du Nguyen, S. Wong, and S. Hamdioui. “Memristive device based circuits for computation-in-memory architectures”. In: *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2019, pp. 1–5.

[60] S. Gupta *et al.* “FELIX: Fast and Energy-Efficient Logic in Memory”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–7.

[61] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli. “The programmable logic-in-memory (PLiM) computer”. In: *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Ieee. 2016, pp. 427–432.

[62] N. TaheriNejad. “Sixor: Single-cycle in-memristor xor”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29.5 (2021), pp. 925–935.

[63] M. Imani *et al.* “DUAL: Acceleration of Clustering Algorithms Using Digital-based Processing In-Memory”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 356–371.

[64] C. Graves *et al.* “In-Memory Computing with Memristor Content Addressable Memories for Pattern Matching”. In: *Advanced Materials* 32.37 (2020), p. 2003437.

[65] D. Fujiki *et al.* “In-memory data parallel processor”. In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 1–14.

[66] L. Xie *et al.* “Scouting logic: A novel memristor-based logic design for resistive computing”. In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2017, pp. 176–181.

[67] A. Shafiee *et al.* “ISAAC: A convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars”. In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 14–26.

[68] I. Giannopoulos *et al.* “In-Memory Database Query”. In: *Advanced Intelligent Systems* 2.12 (2020), p. 2000141.

[69] L. Song *et al.* “Pipelayer: A pipelined ReRAM-based accelerator for deep learning”. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2017, pp. 541–552.

[70] A. Yousefzadeh, J. Stuijt, M. Hijdra, H.-H. Liu, A. Gebregiorgis, A. Singh, S. Hamdioui, and F. Catthoor. “Energy-efficient In-Memory Address Calculation”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 19.4 (2022), pp. 1–16.

[71] J. Yu *et al.* “Time-division multiplexing automata processor”. In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 794–799.

[72] Y.-D. Kim *et al.* “A High-Speed Range-Matching TCAM for Storage-Efficient Packet Classification”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 56.6 (2009), pp. 1221–1230. DOI: 10.1109/TCSI.2008.2008512.

[73] P. Van Mieghem and F. A. Kuipers. “Concepts of exact QoS routing algorithms”. In: *IEEE/ACM Transactions on networking* 12.5 (2004), pp. 851–864.

[74] H. Nili *et al.* “Hardware-intrinsic security primitives enabled by analogue state and nonlinear conductance variations in integrated memristors”. In: *Nature Electronics* 1.3 (2018), pp. 197–202.

[75] B. Cambou *et al.* “Cryptography with analog scheme using memristors”. In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 16.4 (2020), pp. 1–30.

[76] M. Masoumi. “Novel Hybrid CMOS/Memristor Implementation of the AES Algorithm Robust Against Differential Power Analysis Attack”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.7 (2020), pp. 1314–1318. DOI: 10.1109/TCSII.2019.2932337.

[77] S. Muthulakshmi *et al.* “Memristor augmented approximate adders and subtractors for image processing applications: An approach”. In: *AEU-International Journal of Electronics and Communications* 91 (2018), pp. 91–102.

[78] M. Nourazar *et al.* “Memristor-based approximate matrix multiplier”. In: *Analog Integrated Circuits and Signal Processing* 93.2 (2017), pp. 363–373.

[79] I.-A. Fyrigos *et al.* “Memristor Hardware Accelerator of Quantum Computations”. In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2019, pp. 799–802. DOI: 10.1109/ICECS46596.2019.8965109.

[80] M. A. Zidan *et al.* “A general memristor-based partial differential equation solver”. In: *Nature Electronics* 1.7 (2018), pp. 411–420.

[81] Z. Sun *et al.* “In-Memory Eigenvector Computation in Time O (1)”. In: *Advanced Intelligent Systems* 2.8 (2020), p. 2000042.

[82] M. Le Gallo *et al.* “Compressed sensing with approximate message passing using in-memory computing”. In: *IEEE Transactions on Electron Devices* 65.10 (2018), pp. 4304–4312.

[83] R. Cai *et al.* “Memristor-based discrete fourier transform for improving performance and energy efficiency”. In: *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2016, pp. 643–648.

[84] C. Li *et al.* “Analogue signal and image processing with large memristor crossbars”. In: *Nature electronics* 1.1 (2018), pp. 52–59.

[85] H. Tsai *et al.* “Inference of Long-Short Term Memory networks at software-equivalent accuracy using 2.5 M analog Phase Change Memory devices”. In: *2019 Symposium on VLSI Technology*. IEEE. 2019, T82–T83.

[86] C. Du *et al.* “Reservoir computing using dynamic memristors for temporal information processing”. In: *Nature communications* 8.1 (2017), pp. 1–10.

[87] H. A. D. Nguyen *et al.* “A computation-in-memory accelerator based on resistive devices”. In: *Proceedings of the International Symposium on Memory Systems*. 2019, pp. 19–32.

[88] F. Zokaee *et al.* “Finder: Accelerating fm-index-based exact pattern matching in genomic sequences through reram technology”. In: *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2019, pp. 284–295.

[89] Q. Lou *et al.* “Helix: Algorithm/Architecture Co-design for Accelerating Nanopore Genome Base-calling”. In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 2020, pp. 293–304.

[90] T. Shahroodi *et al.* “KrakenOnMem: A Memristor-Augmented HW/SW Framework for Taxonomic Profiling”. In: *Proceedings of the 36th ACM International Conference on Supercomputing*. 2022, pp. 1–14.

[91] L. Song *et al.* “GraphR: Accelerating graph processing using ReRAM”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 531–543.

[92] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu. “Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture”. In: *arXiv preprint arXiv:2105.03814* (2021).

[93] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. “Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology”. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 2017, pp. 273–287.

[94] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay. “ReVAMP: ReRAM based VLIW architecture for in-memory computing”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 782–787.

[95] 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 (2012), pp. 994–1007.

[96] 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*. IEEE. 2012, pp. 392–397.

[97] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. “CACTI 6.0: A tool to model large caches”. In: *HP laboratories* 27 (2009), p. 28.

[98] M. K. F. Lee, Y. Cui, T. Somu, T. Luo, J. Zhou, W. T. Tang, W.-F. Wong, and R. S. M. Goh. “A system-level simulator for RRAM-based neuromorphic computing chips”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 15.4 (2019), pp. 1–24.

[99] D. W. Otter, J. R. Medina, and J. K. Kalita. “A survey of the usages of deep learning for natural language processing”. In: *IEEE transactions on neural networks and learning systems* 32.2 (2020), pp. 604–624.

[100] A. Dhillon and G. K. Verma. “Convolutional neural network: a review of models, methodologies and applications to object detection”. In: *Progress in Artificial Intelligence* 9.2 (2020), pp. 85–112.

[101] W. Wang, Y. Yang, X. Wang, W. Wang, and J. Li. “Development of convolutional neural network and its application in image classification: a survey”. In: *Optical Engineering* 58.4 (2019), p. 040901.

[102] D. Lu and Q. Weng. “A survey of image classification methods and techniques for improving classification performance”. In: *International journal of Remote sensing* 28.5 (2007), pp. 823–870.

[103] W. Wan, R. Kubendran, C. Schaefer, S. B. Eryilmaz, W. Zhang, D. Wu, S. Deiss, P. Raina, H. Qian, B. Gao, *et al.* “A compute-in-memory chip based on resistive random-access memory”. In: *Nature* 608.7923 (2022), pp. 504–512.

[104] P. Yao, H. Wu, B. Gao, J. Tang, Q. Zhang, W. Zhang, J. J. Yang, and H. Qian. “Fully hardware-implemented memristor convolutional neural network”. In: *Nature* 577.7792 (2020), pp. 641–646.

[105] R. Khaddam-Aljameh, M. Stanisavljevic, J. F. Mas, G. Karunaratne, M. Brändli, F. Liu, A. Singh, S. M. Müller, U. Egger, A. Petropoulos, *et al.* “HERMES-core—A 1.59-TOPS/mm 2 PCM on 14-nm CMOS in-memory compute core using 300-ps/LSB linearized CCO-based ADCs”. In: *IEEE Journal of Solid-State Circuits* 57.4 (2022), pp. 1027–1038.

[106] S. Yu, H. Jiang, S. Huang, X. Peng, and A. Lu. “Compute-in-memory chips for deep learning: Recent trends and prospects”. In: *IEEE circuits and systems magazine* 21.3 (2021), pp. 31–56.

[107] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang. “Time: A training-in-memory architecture for memristor-based deep neural networks”. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017, pp. 1–6.

[108] Y. Xi, B. Gao, J. Tang, A. Chen, M.-F. Chang, X. S. Hu, J. Van Der Spiegel, H. Qian, and H. Wu. "In-memory learning with analog resistive switching memory: A review and perspective". In: *Proceedings of the IEEE* 109.1 (2020), pp. 14–42.

[109] H. Tsai, S. Ambrogio, P. Narayanan, R. M. Shelby, and G. W. Burr. "Recent progress in analog memory-based accelerators for deep learning". In: *Journal of Physics D: Applied Physics* 51.28 (2018), p. 283001.

[110] 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: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 27–39.

[111] X. Liu and Z. Zeng. "Memristor crossbar architectures for implementing deep neural networks". In: *Complex & Intelligent Systems* (2022), pp. 1–16.

[112] F. Corinto, A. Ascoli, and S. K. Sung-Mo. "Memristor-based neural circuits". In: *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2013, pp. 417–420.

[113] A. Amirsoleimani, M. Ahmadi, and A. Ahmadi. "STDP-based unsupervised learning of memristive spiking neural network by Morris-Lecar model". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, pp. 3409–3414.

[114] J. Zhang and X. Liao. "Synchronization and chaos in coupled memristor-based FitzHugh-Nagumo circuits with memristor synapse". In: *Aeu-international journal of electronics and communications* 75 (2017), pp. 82–90.

[115] S. Lashkare, S. Chouhan, T. Chavan, A. Bhat, P. Kumbhare, and U. Ganguly. "PCMO RRAM for integrate-and-fire neuron in spiking neural networks". In: *IEEE Electron Device Letters* 39.4 (2018), pp. 484–487.

[116] M. Al-Shedivat, R. Naous, G. Cauwenberghs, and K. N. Salama. "Memristors empower spiking neurons with stochasticity". In: *IEEE journal on Emerging and selected topics in circuits and systems* 5.2 (2015), pp. 242–253.

[117] J. Shamsi, A. Amirsoleimani, S. Mirzakuchaki, and M. Ahmadi. "Modular neuron comprises of memristor-based synapse". In: *Neural Computing and Applications* 28 (2017), pp. 1–11.

[118] A. Mehonic and A. J. Kenyon. "Emulating the electrical activity of the neuron using a silicon oxide RRAM cell". In: *Frontiers in neuroscience* 10 (2016), p. 57.

[119] J.-Q. Yang, R. Wang, Z.-P. Wang, Q.-Y. Ma, J.-Y. Mao, Y. Ren, X. Yang, Y. Zhou, and S.-T. Han. "Leaky integrate-and-fire neurons based on perovskite memristor for spiking neural networks". In: *Nano Energy* 74 (2020), p. 104828.

[120] P. Wijesinghe, A. Ankit, A. Sengupta, and K. Roy. "An all-memristor deep spiking neural computing system: A step toward realizing the low-power stochastic brain". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 2.5 (2018), pp. 345–358.

[121] Z. Zhao, L. Qu, L. Wang, Q. Deng, N. Li, Z. Kang, S. Guo, and W. Xu. “A memristor-based spiking neural network with high scalability and learning efficiency”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.5 (2020), pp. 931–935.

[122] N. Zheng and P. Mazumder. “Online supervised learning for hardware-based multilayer spiking neural networks through the modulation of weight-dependent spike-timing-dependent plasticity”. In: *IEEE transactions on neural networks and learning systems* 29.9 (2017), pp. 4287–4302.

[123] N. Zheng and P. Mazumder. “Learning in memristor crossbar-based spiking neural networks through modulation of weight-dependent spike-timing-dependent plasticity”. In: *IEEE Transactions on Nanotechnology* 17.3 (2018), pp. 520–532.

[124] Y. Zeng, K. Devincentis, Y. Xiao, Z. I. Ferdous, X. Guo, Z. Yan, and Y. Berdichevsky. “A supervised STDP-based training algorithm for living neural networks”. In: *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2018, pp. 1154–1158.

[125] G. Pedretti, V. Milo, S. Ambrogio, R. Carboni, S. Bianchi, A. Calderoni, N. Ramaswamy, A. Spinelli, and D. Ielmini. “Memristive neural network for on-line learning and tracking with brain-inspired spike timing dependent plasticity”. In: *Scientific reports* 7.1 (2017), pp. 1–10.

[126] Y. Nishitani, Y. Kaneko, and M. Ueda. “Supervised learning using spike-timing-dependent plasticity of memristive synapses”. In: *IEEE transactions on neural networks and learning systems* 26.12 (2015), pp. 2999–3008.

[127] C. Sung, H. Hwang, and I. K. Yoo. “Perspective: A review on memristive hardware for neuromorphic computation”. In: *Journal of Applied Physics* 124.15 (2018), p. 151903.

[128] L. Khacef, P. Klein, M. Cartiglia, A. Rubino, G. Indiveri, and E. Chicca. “Spike-based local synaptic plasticity: A survey of computational models and neuromorphic circuits”. In: *arXiv preprint arXiv:2209.15536* (2022).

[129] J. K. Eshraghian, X. Wang, and W. D. Lu. “Memristor-based binarized spiking neural networks: Challenges and applications”. In: *IEEE Nanotechnology Magazine* 16.2 (2022), pp. 14–23.

[130] Y. Long, T. Na, and S. Mukhopadhyay. “ReRAM-based processing-in-memory architecture for recurrent neural network acceleration”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.12 (2018), pp. 2781–2794.

[131] C. Li, Z. Wang, M. Rao, D. Belkin, W. Song, H. Jiang, P. Yan, Y. Li, P. Lin, M. Hu, *et al.* “Long short-term memory networks in memristor crossbar arrays”. In: *Nature Machine Intelligence* 1.1 (2019), pp. 49–57.

[132] K. Adam, K. Smagulova, and A. P. James. “Memristive LSTM network hardware architecture for time-series predictive modeling problems”. In: *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE. 2018, pp. 459–462.

[133] K. Smagulova, K. Adam, O. Krestinskaya, and A. P. James. “Design of cmos-memristor circuits for lstm architecture”. In: *2018 IEEE international conference on electron devices and solid state circuits (EDSSC)*. IEEE. 2018, pp. 1–2.

[134] K. Smagulova, O. Krestinskaya, and A. P. James. “A memristor-based long short term memory circuit”. In: *Analog Integrated Circuits and Signal Processing* 95 (2018), pp. 467–472.

[135] K. Smagulova and A. P. James. “A survey on LSTM memristive neural network architectures and applications”. In: *The European Physical Journal Special Topics* 228.10 (2019), pp. 2313–2324.

[136] C. Yuan and S. S. Agaian. “A comprehensive review of Binary Neural Network”. In: *arXiv preprint arXiv:2110.06804* (2021).

[137] J. Chen, S. Wen, K. Shi, and Y. Yang. “Highly parallelized memristive binary neural network”. In: *Neural Networks* 144 (2021), pp. 565–572.

[138] T. Tang, L. Xia, B. Li, Y. Wang, and H. Yang. “Binary convolutional neural network on RRAM”. In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2017, pp. 782–787.

[139] L. Huang, J. Diao, H. Nie, W. Wang, Z. Li, Q. Li, and H. Liu. “Memristor based binary convolutional neural network architecture with configurable neurons”. In: *Frontiers in neuroscience* 15 (2021), p. 328.

[140] Y.-F. Qin, R. Kuang, X.-D. Huang, Y. Li, J. Chen, and X.-S. Miao. “Design of high robustness BNN inference accelerator based on binary memristors”. In: *IEEE Transactions on Electron Devices* 67.8 (2020), pp. 3435–3441.

[141] Y. Zhao, Y. Wang, R. Wang, Y. Rong, and X. Jiang. “A Highly Robust Binary Neural Network Inference Accelerator Based on Binary Memristors”. In: *Electronics* 10.21 (2021), p. 2600.

[142] Y. Halawani, B. Mohammad, M. A. Lebdeh, M. Al-Qutayri, and S. F. Al-Sarawi. “ReRAM-based in-memory computing for search engine and neural network applications”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 388–397.

[143] I. Pitas. *Graph-based social media analysis*. Vol. 39. CRC Press, 2016.

[144] X.-M. Zhang *et al.* “Graph neural networks and their current applications in bioinformatics”. In: *Frontiers in genetics* 12 (2021).

[145] H. Peng *et al.* “Spatial temporal incidence dynamic graph neural networks for traffic flow forecasting”. In: *Information Sciences* 521 (2020), pp. 277–290.

[146] X. Dong *et al.* “Graph signal processing for machine learning: A review and new perspectives”. In: *IEEE Signal processing magazine* 37.6 (2020), pp. 117–127.

[147] M. Zhang *et al.* “GraphP: Reducing communication for PIM-based graph processing with efficient data partition”. In: *HPCA*. IEEE. 2018, pp. 544–557.

[148] J. Lin *et al.* “Overcoming the Memory Hierarchy Inefficiencies in Graph Processing Applications”. In: *IEEE/ACM ICCAD*. IEEE. 2021, pp. 1–9.

[149] S. Rahman *et al.* “Graphpulse: An event-driven hardware accelerator for asynchronous graph processing”. In: *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020.

[150] J. Ahn *et al.* “A scalable processing-in-memory accelerator for parallel graph processing”. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 2015, pp. 105–117.

[151] G. Dai *et al.* “Graphh: A processing-in-memory architecture for large-scale graph processing”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.4 (2018), pp. 640–653.

[152] Y. Zhuo *et al.* “Graphq: Scalable PIM-based graph processing”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 712–725.

[153] M. Zhou *et al.* “Gram: graph processing in a ReRAM-based computational memory”. In: *IEEE Asia and South Pacific Design Automation Conference*. 2019.

[154] G. Dai *et al.* “GraphSAR: A sparsity-aware processing-in-memory architecture for large-scale graph processing on ReRAMs”. In: *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 2019, pp. 120–126.

[155] M. Alser, Z. Bingöl, D. S. Cali, J. Kim, S. Ghose, C. Alkan, and O. Mutlu. “Accelerating genome analysis: A primer on an ongoing journey”. In: *IEEE Micro* 40.5 (2020), pp. 65–75.

[156] F. Syed, H. Grunenwald, and N. Caruccio. *Next-generation sequencing library preparation: simultaneous fragmentation and tagging using in vitro transposition*. 2009.

[157] J. Shendure and H. Ji. “Next-generation DNA sequencing”. In: *Nature biotechnology* 26.10 (2008), pp. 1135–1145.

[158] S. E. Levy and R. M. Myers. “Advancements in next-generation sequencing”. In: *Annual review of genomics and human genetics* 17 (2016), pp. 95–115.

[159] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, *et al.* “Personalized copy number and segmental duplication maps using next-generation sequencing”. In: *Nature genetics* 41.10 (2009), pp. 1061–1067.

[160] H. Li. “Minimap2: pairwise alignment for nucleotide sequences”. In: *Bioinformatics* 34.18 (2018), pp. 3094–3100.

[161] W. Huangfu, S. Li, X. Hu, and Y. Xie. “RADAR: A 3D-ReRAM based DNA alignment accelerator architecture”. In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6.

[162] S. K. Khatamifard, Z. Chowdhury, N. Pande, M. Razaviyayn, C. Kim, and U. R. Karpuzcu. "GeNVoM: Read mapping near non-volatile memory". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.6 (2021), pp. 3482–3496.

[163] F. Zokaee, H. R. Zarandi, and L. Jiang. "Aligner: A process-in-memory architecture for short read alignment in rerams". In: *IEEE Computer Architecture Letters* 17.2 (2018), pp. 237–240.

[164] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy. "GenAx: A genome sequencing accelerator". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 69–82.

[165] A. Zeni, G. Guidi, M. Ellis, N. Ding, M. D. Santambrogio, S. Hofmeyr, A. Buluç, L. Oliker, and K. Yelick. "Logan: High-performance gpu-based x-drop long-read alignment". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2020, pp. 462–471.

[166] N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels, and Z. Al-Ars. "GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data". In: *BMC bioinformatics* 20 (2019), pp. 1–20.

[167] Y.-L. Chen, B.-Y. Chang, C.-H. Yang, and T.-D. Chiueh. "A high-throughput FPGA accelerator for short-read mapping of the whole human genome". In: *IEEE Transactions on Parallel and Distributed Systems* 32.6 (2021), pp. 1465–1478.

[168] A. Nag, C. Ramachandra, R. Balasubramonian, R. Stutsman, E. Giacomin, H. Kambalasubramanyam, and P.-E. Gaillardon. "Gencache: Leveraging in-cache operators for efficient sequence alignment". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 334–346.

[169] M. Alser, T. Shahroodi, J. Gómez-Luna, C. Alkan, and O. Mutlu. "SneakySnake: a fast and accurate universal genome pre-alignment filter for CPUs, GPUs and FPGAs". In: *Bioinformatics* 36.22-23 (2020), pp. 5282–5290.

[170] Z. Bingöl, M. Alser, O. Mutlu, O. Ozturk, and C. Alkan. "Gatekeeper-gpu: Fast and accurate pre-alignment filtering in short read mapping". In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2021, pp. 209–209.

[171] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu. "Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping". In: *Bioinformatics* 31.10 (2015), pp. 1553–1560.

[172] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, *et al.* "GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 951–966.

[173] Y. Turakhia, G. Bejerano, and W. J. Dally. "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly". In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 199–213.

[174] C. Firtina, K. Pillai, G. S. Kalsi, B. Suresh, D. S. Cali, J. Kim, T. Shahroodi, M. B. Cavlak, J. Lindegger, M. Alser, *et al.* "ApHMM: Accelerating profile hidden markov models for fast and energy-efficient genome analysis". In: *arXiv preprint arXiv:2207.09765* (2022).

[175] T. Shahroodi, M. Zahedi, A. Singh, S. Wong, and S. Hamdioui. "KrakenOnMem: a memristor-augmented HW/SW framework for taxonomic profiling". In: *Proceedings of the 36th ACM International Conference on Supercomputing*. 2022, pp. 1–14.

[176] C. Firtina, J. Park, M. Alser, J. S. Kim, D. S. Cali, T. Shahroodi, N. M. Ghiasi, G. Singh, K. Kanellopoulos, C. Alkan, *et al.* "BLEND: a fast, memory-efficient and accurate mechanism to find fuzzy seed matches in genome analysis". In: *NAR Genomics and Bioinformatics* 5.1 (2023), lqad004.

[177] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. "Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[178] O. Golonzka, U. Arslan, P. Bai, M. Bohr, O. Baykan, Y. Chang, A. Chaudhari, A. Chen, J. Clarke, C. Connor, *et al.* "Non-volatile RRAM embedded into 22FFL FinFET technology". In: *2019 Symposium on VLSI Technology*. IEEE. 2019, T230–T231.

[179] R. Dittmann, S. Menzel, and R. Waser. "Nanoionic memristive phenomena in metal oxides: the valence change mechanism". In: *Advances in Physics* 70.2 (2021), pp. 155–349.

[180] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. "Xnor-net: Imagenet classification using binary convolutional neural networks". In: *European conference on computer vision*. Springer. 2016, pp. 525–542.

[181] *BCIM simulation platform*. URL: https://github.com/mahdi-zahedi/BCIM-Binary-Neural-Network-using-Computation-in-Memory-.git.

[182] M. Courbariaux, Y. Bengio, and J.-P. David. "Binaryconnect: Training deep neural networks with binary weights during propagations". In: *Advances in neural information processing systems* 28 (2015).

[183] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei. "FP-BNN: Binarized neural network on FPGA". In: *Neurocomputing* 275 (2018), pp. 1072–1086. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2017.09.046. URL: https://www.sciencedirect.com/science/article/pii/S0925231217315655.

[184] C. Fu, S. Zhu, H. Su, C.-E. Lee, and J. Zhao. "Towards fast and energy-efficient binarized neural network inference on fpga". In: *arXiv preprint arXiv:1810.02068* (2018).

[185] H. Yang, M. Fritzsche, C. Bartz, and C. Meinel. “Bmxnet: An open-source binary neural network implementation based on mxnet”. In: *Proceedings of the 25th ACM international conference on Multimedia*. 2017, pp. 1209–1212.

[186] M. Hu, C. E. Graves, C. Li, Y. Li, N. Ge, E. Montgomery, N. Davila, H. Jiang, R. S. Williams, J. J. Yang, *et al.* “Memristor-based analog computation and neural network classification with a dot product engine”. In: *Advanced Materials* 30.9 (2018), p. 1705914.

[187] Y. Kim, W. H. Jeong, S. B. Tran, H. C. Woo, J. Kim, C. S. Hwang, K.-S. Min, and B. J. Choi. “Memristor crossbar array for binarized neural networks”. In: *AIP Advances* 9.4 (2019), p. 045131.

[188] D. Ahn, H. Oh, H. Kim, Y. Kim, and J.-J. Kim. “Maximizing Parallel Activation of Word-Lines in MRAM-Based Binary Neural Network Accelerators”. In: *IEEE Access* 9 (2021), pp. 141961–141969.

[189] P. Narayanan, S. Ambrogio, A. Okazaki, K. Hosokawa, H. Tsai, A. Nomura, T. Yasuda, C. Mackin, S. Lewis, A. Friz, *et al.* “Fully on-chip MAC at 14nm enabled by accurate row-wise programming of PCM-based weights and parallel vector-transport in duration-format”. In: *2021 Symposium on VLSI Technology*. IEEE. 2021, pp. 1–2.

[190] G. Karunaratne, M. Le Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian. “In-memory hyperdimensional computing”. In: *Nature Electronics* 3.6 (2020), pp. 327–337.

[191] J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[192] K. Fleck *et al.* “Energy dissipation during pulsed switching of strontium-titanate based resistive switching memory devices”. In: *ESSDERC*. IEEE. 2016, pp. 160–163.

[193] L. Kull *et al.* “A 3.1 mW 8b 1.2 GS/s single-channel asynchronous SAR ADC with alternate comparators for enhanced speed in 32 nm digital SOI CMOS”. In: *IEEE JSSC* 48.12 (2013), pp. 3049–3058.

[194] M. Saberi *et al.* “Analysis of power consumption and linearity in capacitive digital-to-analog converters used in successive approximation ADCs”. In: *IEEE TCAS I: Regular Papers* 58.8 (2011), pp. 1736–1748.

[195] R. Kobus, J. M. Abuın, A. Müller, S. L. Hellmann, J. C. Pichel, T. F. Pena, A. Hildebrandt, T. Hankeln, and B. Schmidt. “A big data approach to metagenomics for all-food-sequencing”. In: *BMC bioinformatics* 21.1 (2020), pp. 1–15.

[196] F. Ripp, C. F. Krombholz, Y. Liu, M. Weber, A. Schäfer, B. Schmidt, R. Köppel, and T. Hankeln. “All-Food-Seq (AFS): a quantifiable screen for species in biological samples by deep DNA sequencing”. In: *BMC genomics* 15.1 (2014), pp. 1–11.

[197] Wetterstrand KA. *DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP)*. https://www.genome.gov/sequencingcostsdata.

[198] Barba, M, Czosnek, H and Hadidi, A. *Cost in US Dollars per Raw Megabase of DNA Sequence*. https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data.

[199] D. E. Wood, J. Lu, and B. Langmead. "Improved Metagenomic Analysis with Kraken 2". In: *Genome biology* 20.1 (2019), pp. 1–13.

[200] A. E. Pérez-Cobas, L. Gomez-Valero, and C. Buchrieser. "Metagenomic approaches in microbial ecology: an update on whole-genome and marker gene sequencing analyses". In: *Microbial genomics* 6.8 (2020).

[201] A. B. McIntyre, R. Ounit, E. Afshinnekoo, R. J. Prill, E. Hénaff, N. Alexander, S. S. Minot, D. Danko, J. Foox, S. Ahsanuddin, *et al.* "Comprehensive benchmarking and ensemble approaches for metagenomic classifiers". In: *Genome biology* 18.1 (2017), pp. 1–19.

[202] F. P. Breitwieser, J. Lu, and S. L. Salzberg. "A Review of Methods and Databases for Metagenomic Classification and Assembly". In: *Briefings in bioinformatics* 20.4 (2019), pp. 1125–1136.

[203] L. Ge and K. K. Parhi. "Classification using Hyperdimensional Computing: A Review". In: *IEEE Circuits and Systems Magazine* 20.2 (2020), pp. 30–47.

[204] P. Kanerva. "Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors". In: *Cognitive computation* 1.2 (2009), pp. 139–159.

[205] R. W. Gayler. "Vector symbolic architectures answer Jackendoff's challenges for cognitive neuroscience". In: *arXiv preprint cs/0412059* (2004).

[206] F. R. Najafabadi, A. Rahimi, P. Kanerva, and J. M. Rabaey. "Hyperdimensional computing for text classification". In: *Design, automation test in Europe conference exhibition (DATE), University Booth*. 2016, pp. 1–1.

[207] A. Rahimi, S. Benatti, P. Kanerva, L. Benini, and J. M. Rabaey. "Hyperdimensional biosignal processing: A case study for EMG-based hand gesture recognition". In: *IEEE International Conference on Rebooting Computing (ICRC)*. 2016, pp. 1–8.

[208] P. Kanerva, J. Kristoferson, and A. Holst. "Random indexing of text samples for latent semantic analysis". In: *Proceedings of the Annual Meeting of the Cognitive Science Society*. Vol. 22. 22. 2000.

[209] D. Kleyko, A. Rahimi, D. A. Rachkovskij, E. Osipov, and J. M. Rabaey. "Classification and Recall with Binary Hyperdimensional Computing: Tradeoffs in Choice of Density and Mapping Characteristics". In: *IEEE transactions on neural networks and learning systems* 29.12 (2018), pp. 5880–5898.

[210] A. Thomas, S. Dasgupta, and T. Rosing. "A theoretical perspective on hyperdimensional computing". In: *Journal of Artificial Intelligence Research* 72 (2021), pp. 215–249.

[211] D. Kleyko, D. Rachkovskij, E. Osipov, and A. Rahimi. “A Survey on Hyperdimensional Computing Aka Vector Symbolic Architectures, Part II: Applications, Cognitive Models, and Challenges”. In: *ACM Comput. Surv.* 55.9 (Jan. 2023). ISSN: 0360-0300. DOI: 10.1145/3558000. URL: https://doi.org/10.1145/3558000.

[212] N. LaPierre, S. Mangul, M. Alser, I. Mandric, N. C. Wu, D. Koslicki, and E. Eskin. “MiCoP: Microbial Community Profiling Method for Detecting Viral and Fungal Organisms in Metagenomic Samples”. In: *BMC genomics* 20.5 (2019), pp. 1–10.

[213] N. LaPierre, M. Alser, E. Eskin, D. Koslicki, and S. Mangul. “Metalign: Efficient Alignment-Based Metagenomic Profiling via Containment Min Hash”. In: *Genome biology* 21.1 (2020), pp. 1–15.

[214] J. Lu, F. P. Breitwieser, P. Thielen, and S. L. Salzberg. “Bracken: Estimating Species Abundance in Metagenomics Data”. In: *PeerJ Computer Science* 3 (2017), e104.

[215] R. Ounit, S. Wanamaker, T. J. Close, and S. Lonardi. “CLARK: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers”. In: *BMC genomics* 16.1 (2015), pp. 1–13.

[216] D. Fujiki, S. Mahlke, and R. Das. “Duality cache for data parallel acceleration”. In: *Proceedings of the 46th International Symposium on Computer Architecture*. 2019, pp. 397–410.

[217] I. Merelli, L. Morganti, E. Corni, C. Pellegrino, D. Cesini, L. Roverelli, G. Zereik, and D. D’Agostino. “Low-power portable devices for metagenomics analysis: Fog computing makes bioinformatics ready for the Internet of Things”. In: *Future Generation Computer Systems* 88 (2018), pp. 467–478.

[218] D. D’Agostino, L. Morganti, E. Corni, D. Cesini, and I. Merelli. “Combining edge and cloud computing for low-power, cost-effective metagenomics analysis”. In: *Future Generation Computer Systems* 90 (2019), pp. 79–85.

[219] Ö. Eyice, M. Namura, Y. Chen, A. Mead, S. Samavedam, and H. Schäfer. “SIP metagenomics identifies uncultivated Methylophilaceae as dimethylsulphide degrading bacteria in soil and lake sediment”. In: *The ISME journal* 9.11 (2015), pp. 2336–2348.

[220] R. van Duijnen. “ISA and Hardware Design for a Pipelined CIM-tile”. In: (2020).

[221] A. BanaGozar, K. Vadivel, S. Stuijk, H. Corporaal, S. Wong, M. A. Lebdeh, J. Yu, and S. Hamdioui. “Cim-sim: computation in memory simuiator”. In: *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*. 2019, pp. 1–4.

[222] K. Vadivel, L. Chelini, A. BanaGozar, G. Singh, S. Corda, R. Jordans, and H. Corporaal. “Tdo-cim: Transparent detection and offloading for computation in-memory”. In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 1602–1605.

[223] A. Hardtdegen, C. La Torre, F. Cüppers, S. Menzel, R. Waser, and S. Hoffmann-Eifert. “Improved switching stability and the effect of an internal series resistor in HfO 2/TiO x bilayer ReRAM cells”. In: *IEEE transactions on electron devices* 65.8 (2018), pp. 3229–3236.

[224] M. Le Gallo, S. Nandakumar, L. Ciric, I. Boybat, R. Khaddam-Aljameh, C. Mackin, and A. Sebastian. “Precision of bit slicing with in-memory computing based on analog phase-change memory crossbars”. In: *Neuromorphic Computing and Engineering* 2.1 (2022), p. 014009.

[225] T. Chou, W. Tang, J. Botimer, and Z. Zhang. “CASCADE: Connecting RRAMs to Extend Analog Dataflow In An End-To-End In-Memory Processing Paradigm”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 114–125.

[226] F. Cai, J. M. Correll, S. H. Lee, Y. Lim, V. Bothra, Z. Zhang, M. P. Flynn, and W. D. Lu. “A fully integrated reprogrammable memristor–CMOS system for efficient multiply–accumulate operations”. In: *Nature Electronics* 2.7 (2019), pp. 290–299.

[227] P. Yao, H. Wu, B. Gao, J. Tang, Q. Zhang, W. Zhang, J. J. Yang, and H. Qian. “Fully hardware-implemented memristor convolutional neural network”. In: *Nature* 577.7792 (2020), pp. 641–646.

[228] M. Imani, S. Gupta, Y. Kim, and T. Rosing. “FloatPIM: In-memory Acceleration of Deep Neural Network Training with High Precision”. In: *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2019, pp. 802–815.

[229] *Memristor-CIM-tile-architecture-TUDeflt*. URL: https://github.com/mahdi-zahedi/Memristor-CIM-tile-architecture-TUDeflt-.

[230] S. Rashidi, M. Jalili, and H. Sarbazi-Azad. “A Survey on PCM Lifetime Enhancement Schemes”. In: *ACM Comput. Surv.* 52.4 (Aug. 2019), 76:1–76:38. ISSN: 0360-0300. DOI: 10.1145/3332257. URL: http://doi.acm.org/10.1145/3332257.

CURRICULUM VITÆ

Mahdi ZAHEDI

22-03-1993 Born in Tehran, Iran.

EDUCATION

2019-2023 PhD. degree in Computer Engineering
Delft University of Technology
Thesis: Computation-in-Memory from Application-Specific to Programmable Designs
Promotors: Prof. Said Hamdioui, Dr. Stephan Wong

2015-2018 M.Sc. degree in Electrical Engineering, Digital System Design
University of Tehran
Thesis: Improving the performance of approximate digital systems by considering predictor components
Supervisors: Prof. Ali Afzali-Kusha, Dr. Mehdi Kamal

2011-2015 B.Sc. degree in Electrical Engineering, Electronics
University of Shahid Beheshti

LIST OF PUBLICATIONS

11. **M. Zahedi**, G. Custers, T. Shahroodi, G. Gaydadjiev, S. Wong and S. Hamdioui, "SparseMEM: Energy-efficient Design for In-memory Sparse-based Graph Processing," 2023 Design, Automation & Test in Europe Conference Exhibition (DATE), Antwerp, Belgium, 2023, pp. 1-6, doi: 10.23919/DATE56975.2023.10137303.

10. **M. Zahedi**, T. Shahroodi, S. Wong and S. Hamdioui, "Efficient Signed Arithmetic Multiplication on Memristor-Based Crossbar," in IEEE Access, vol. 11, pp. 33964-33978, 2023, doi: 10.1109/ACCESS.2023.3263259.

9. **M. Zahedi**, T. Shahroodi, C. Escuin, G. Gaydadjiev, S. Wong, and S. Hamdioui. “BCIM: Efficient Implementation of Binary Neural Network Based on Computation in Memory”. In: submitted to IEEE Transactions on Emerging Topics in Computing (TETC) 11 (2023), pp. 33964–33978.

8. **M. Zahedi**, T. Shahroodi, G. Custers, A. Singh, S. Wong and S. Hamdioui, "System Design for Computation-in-Memory: From Primitive to Complex Functions," 2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC), Patras, Greece, 2022, pp. 1-6, doi: 10.1109/VLSI-SoC54400.2022.9939571.

7. **Zahedi, M.**, Lebdeh, M.A., Bengel, C., Wouters, D., Menzel, S., Le Gallo, M., Sebastian, A., Wong, S. and Hamdioui, S., 2022. MNEMOSENE: Tile Architecture and Simulator for Memristor-based Computation-in-memory. ACM Journal on Emerging Technologies in Computing Systems (JETC), 18(3), pp.1-24.

6. **M. Zahedi**, R. van Duijnen, S. Wong and S. Hamdioui, “Tile Architecture and Hardware Implementation for Computation-in-Memory,” 2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Tampa, FL, USA, 2021, pp. 108-113, doi: 10.1109/ISVLSI51109.2021.00030.

5. **M. Zahedi**, M. Mayahinia, M. Abu Lebdeh, S. Wong and S. Hamdioui, "Efficient Organization of Digital Periphery to Support Integer Datatype for Memristor-Based CIM," 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Limassol, Cyprus, 2020, pp. 216-221, doi: 10.1109/ISVLSI49217.2020.00047.

4. Shahroodi, T., **Zahedi, M.**, Firtina, C., Alser, M., Wong, S., Mutlu, O. and Hamdioui, S., 2022. Demeter: A fast and energy-efficient food profiler using hyperdimensional computing in memory. IEEE Access, 10, pp.82493-82510.

3. Shahroodi, T., **Zahedi, M.**, Singh, A., Wong, S. and Hamdioui, S., 2022, June. KrakenOnMem: a memristor-augmented HW/SW framework for taxonomic profiling. In Proceedings of the 36th ACM International Conference on Supercomputing (pp. 1-14).

2. Singh, A., **Zahedi, M.**, Shahroodi, T., Gupta, M., Gebregiorgis, A., Komalan, M., Joshi, R.V., Catthoor, F., Bishnoi, R. and Hamdioui, S., 2022, June. Cim-based robust logic accelerator using 28 nm stt-mram characterization chip tape-out. In 2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS) (pp. 451-454). IEEE.

1. Shahroodi, T., Cardoso, R., **Zahedi, M.**, Wong, S., Bosio, A., O'Connor, I. and Hamdioui, S., 2023, April. Lightspeed Binary Neural Networks using Optical Phase-Change Materials. In 2023 Design, Automation & Test in Europe Conference Exhibition (DATE) (pp. 1-2). IEEE.