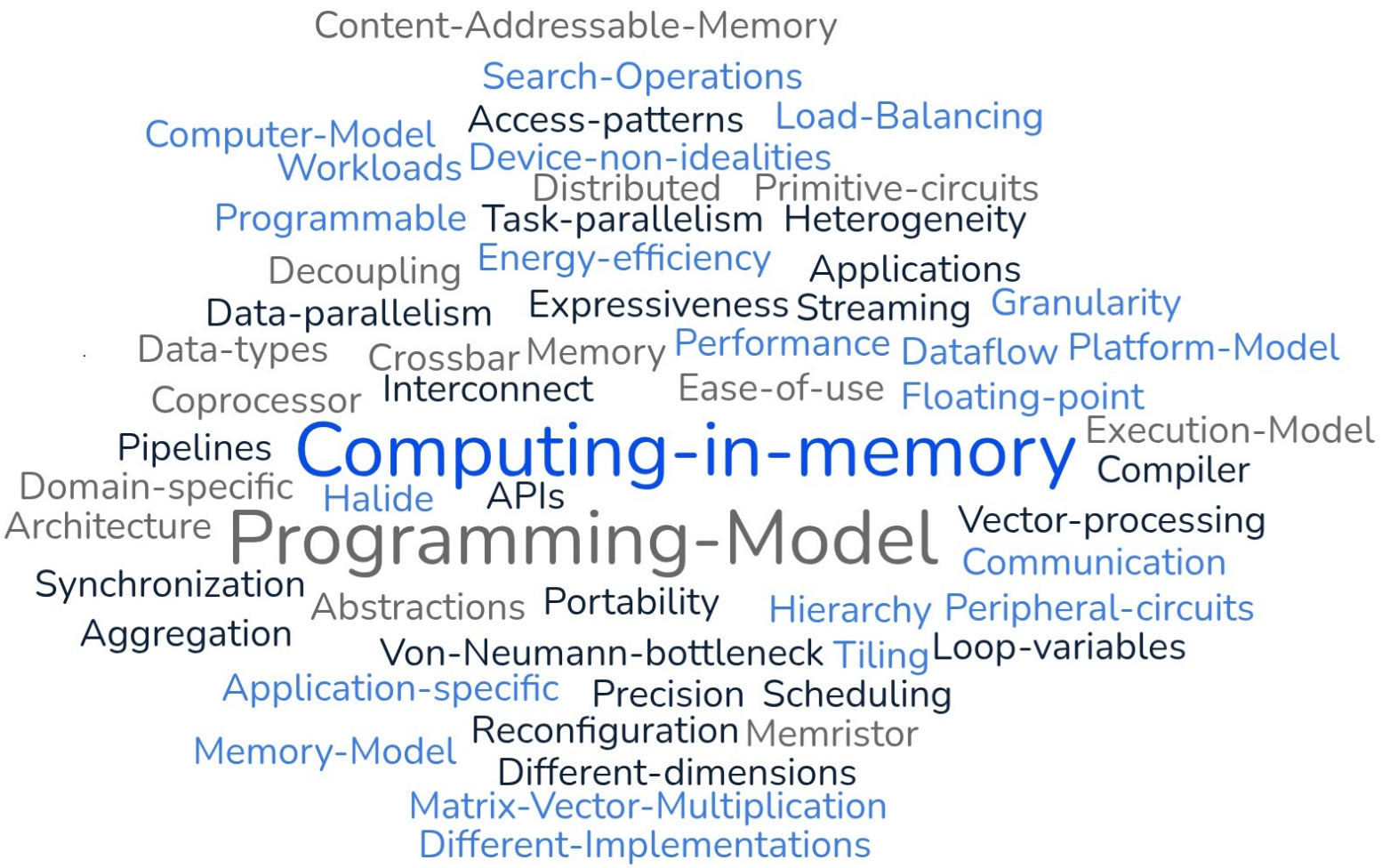
Towards Programmable Computing-in-Memory Accelerators

Design of a general purpose programming model

Thomas Maliappis



TU Delft

Towards Programmable Computing-in-Memory Accelerators

Design of a general purpose programming model

by

Thomas Maliappis

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday December 9, 2025 at 1:15 PM.

Student number:	5336910	
Project duration:	September 1, 2024 – December 9, 2025	
Thesis committee:	Prof.dr.ir Georgi Gaydadjiev,	TU Delft, supervisor
	Prof. dr. K.G. Langendoen,	TU Delft external examiner

Cover: Word cloud image generated at https://www.simplewordcloud.com/
Language Assistance: This paper was edited for grammar and clarity using Grammarly and the AI tool ChatGPT.
An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Acknowledgments

I am sincerely grateful to all my supervisors **Prof Georgi Gaydadjiev**, **Mahmood Naderan-Tahan** and **Theofilos Spyrou** all of which have profoundly shaped the quality and direction of this research. I would like to thank **Georgi Gaydadjiev** for entrusting me with such a challenging topic and for his invaluable guidance during times of uncertainty. His continuous support and the insightful resources he provided demonstrated a deep commitment not only to the success of the research but also to ensuring that his researchers develop a thorough understanding of the subject matter. **Mahmood Naderan-Tahan**, who was my daily supervisor and would always take the time to understand the different topics and perspectives I presented, and they consistently provided thoughtful guidance that helped move the work forward. **Theofilos Spyrous**, whose constructive feedback not only improved the quality of the thesis but also challenged and expanded my understanding of the field.

Next, I would like to thank my friends and family for their unwavering support throughout these five years abroad. I am deeply grateful to my parents for always believing in me, even during times when I struggled to believe in myself. To my siblings, for whom I strive every day to be a role model. To my friends—**Davis, Giulio, Mim, Tim, Martin, and Artjom**—thank you for making every week of work worthwhile through your constant companionship and the unforgettable memories we created together. A special thanks to **Theodoros**, who not only rekindled my passion for computer engineering but also reminded me of the joy and purpose behind what I do. I couldn't have done it without all of you.

Thomas Maliappis
Delft, December 2025

Contents

Alphabetical List of Terms and Definitions

CIM application Any workload for which a dedicated CIM accelerator exists, but which is more complex than the elementary operations natively supported by CIM memory arrays . 7

CIM chip A set of CIM cores in a hierarchical CIM organization . 37

CIM core A set of CIM macros hierarchical CIM organization . 37

CIM macro A CIM primitive accompanied by peripheral circuitry . 14

CIM primitive An analog memory array structure for CIM computations, defined by Reis et al. [1] . 13

computer model An abstraction over a computing system that defines both its underlying structure and its behavior. It consist of: A platform model, an execution model and a memory model . 23

DSL A Domain-Specific Language is the realization of a programming model through the addition of syntax, domain-specific semantics, and programming paradigms . 10

execution model Defines how instructions are scheduled and executed on a computer's hardware, enabling programmers to predict a program's behavior by mentally simulating its execution . 23

general purpose programming model A programming model that can be applied intuitively across a variety of application domains. Unlike its ideal counterpart, it does not claim to unify all possible hardware implementations or guarantee performance . 9

ideal programming model A programming model that unifies diverse application domains and hardware implementations, while delivering high performance comparable to code optimized for a specific architecture and providing an intuitive programming experience . 8

memory model Describes the different memory regions within the system and the dataflow of execution such as the inter-thread interaction within memory and sharing of variables . 23

meta-abstraction An abstraction of an abstraction, designed to encapsulate common patterns or features used by different programming model abstractions . 9

platform model Abstracts away the underlying hardware details and variations across different implementations, presenting a unified and consistent architecture . 23

programming model A set of high-level abstractions built on top of a programmable computer model, designed to expose specific computational or architectural properties to the programmer . 9

Stream A memory object type defined in our proposed programming model. A stream is a handle to an ordered sequence of a single data type that may not yet be available and can potentially be of unbounded length. Unlike tensors, streams are specifically intended to feed preloaded macros by continuously streaming data through their input buffers . 40

symbolic variable A placeholder that binds to an iteration dimension or an index position of a function, as introduced in Halide Section 2.3.4. It does not hold a numerical value at runtime . 41

Tensor A memory object type defined in our proposed programming model. A tensor is an ordered, immutable, multidimensional array of fixed dimensions containing elements of a single data type, with all values fully computed and available. Tensors can be distributed across the different macros of the accelerator and stored within their CIM primitives. . 40

1 Introduction

For a very long period of time, computing could meet the increasing demands of different applications due to the continued downscaling of transistors, which allowed data to be processed at a higher frequency. In the early 2000s, predictions about the physical limits and rising costs of continued downscaling prompted researchers to adopt alternative techniques to sustain performance improvements beyond frequency scaling. Among these, the most prevalent technique was the extraction and utilization of parallelism, which successfully extended performance scaling for more than a decade but has since begun to stagnate. Today, experts agree that specialized complementary hardware is crucial for further advancements. Computing-in-memory (CIM) accelerators are gaining traction as an innovative solution to the problems conventional computing is facing. While most CIM research is directed towards device, circuit, and architectural level challenges, it is also important to consider the challenges at the programming level. In this chapter, we first discuss the motivation behind CIM accelerators and why developing a programming model for them is essential. Next, we provide an overview of the challenges associated with developing a dedicated programming model for this emerging technology. Finally, we will outline the research direction of this thesis.

1.1. Motivation

Advances in computing have long been driven by the increasing demands of modern applications. The question that computer engineers must continually answer is how to innovate, optimize and scale technology in an effort to achieve greater performance while reducing energy consumption, area requirements and economic cost all at the same time. This exponential wave of innovation is not fueled by entirely new concepts, but from the continued refinement of Von Neumann computers that have been pushed forward with each generation of computers. Advancements such as transistor scaling, the memory hierarchy and later the integration of parallelism have helped mask the bottleneck that is the frequent and slow transfer of data and instructions between the processor and memory. As these approaches have begun to hit their physical and practical limits, researchers have considered diverging from the Von Neumann model.

A novel computing paradigm called Computing in Memory (CIM) performs computations directly in memory, drastically reducing data movement and resulting in lower power consumption and higher performance. Hence, CIM accelerators have become widely adopted in various data-intensive applications to complement Von Neumann architectures. CIM introduces a shift in how we perform computation in hardware and significant research is being aimed to improve the technology at the device, circuit and architectural levels.

In order to make CIM available and utilize it effectively we need to devote research on CIM at the programming level as well. After all, we have built most of our existing software tools and our general understanding of computing around the Von Neumann model and as we move away from it, we must question if a new programming model is necessary. If this is the case, we risk designing a coprocessor whose theoretical benefits remain unrealized in practice, as an inadequate programming model could render interaction with the hardware cumbersome, inefficient, or overly restrictive. Hence, in this thesis, we lay the groundwork for the design of future CIM programming models.

In the following sections, we will first provide a detailed overview of the evolution of Von Neumann computers from the inception of the Von Neumann model to current predictions about its future. Specifically, we will mention its limitations and the techniques developed to extend and adapt conventional computers in response to the growing demands of modern applications. Subsequently, we will also introduce the concept of CIM, highlighting its potential benefits to conventional computers as well as the programming challenges it currently presents.

1.1.1. Von Neumann architecture

Ever since its inception in 1945 [2], the Von Neumann model has defined digital computers. Most importantly, it specifies that a computer architecture that consists of three components, shown in Figure 1.1a: a central processing unit (CPU) for performing computations, a memory unit for storing both program instructions/data, and input/output ports for communicating with external devices. The CPU itself is composed of an arithmetic logic unit (ALU), which performs arithmetic and logic operations, and a control unit, which coordinates the execution of instructions. During execution, instructions and data are transferred from memory or an input device into the CPU, and once the results are computed, they are written back to memory or an output device. As a result, two primary types of operations can occur during execution: data movement and computation.

In Figure 1.1a, we illustrate the main bottleneck of the Von Neumann model, which is the communication between the processor and memory, commonly referred to as the ”memory wall”. The memory wall arises from the growing performance gap between the processor and memory, as shown in Figure 1.1b. This gap exists because processor frequencies have improved at a much faster rate than memory latency and bandwidth, causing processors to spend an increasing proportion of time waiting for data or instructions from memory.

To help programmers reason about sequential computers, the Von Neumann model was abstracted into the Random Access Machine (RAM) computation model [3, p.44], which idealizes execution by assuming an unbounded memory with uniform access time. In this model, a single instruction is fetched and executed at constant intervals. The RAM model’s behavior is so deeply ingrained in programming that it is often regarded as the default framework for reasoning about computation.

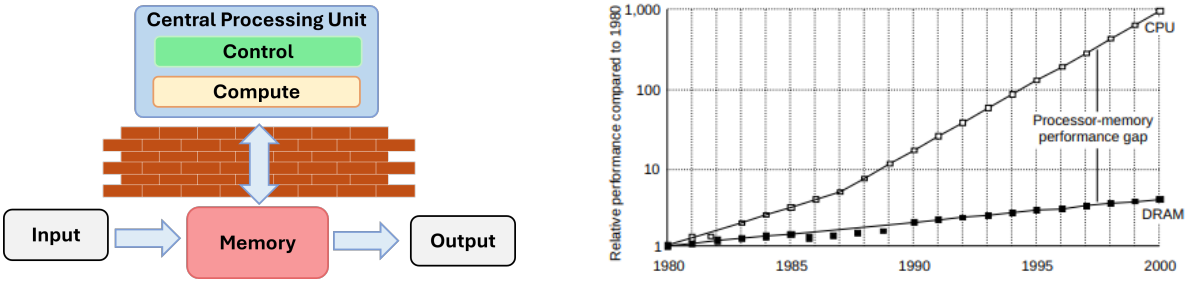


Figure 1.1: (a) The Von Neumann model, its components and the memory wall visualized. (b) The growing Processor-Memory Performance gap that is the cause behind the memory wall[4].

The concept of the memory hierarchy, first proposed in 1946 [5] and illustrated in Figure 1.2a, augmented the Von Neumann model by mitigating the growing performance gap between the processor and memory. As we move from the bottom to the top of the hierarchy, each level trades storage capacity for faster access times by employing more expensive memory technologies. During execution, the hierarchy enables caching mechanisms that exploit temporal and spatial locality. Specifically, if certain data or instructions are frequently accessed, or if nearby accesses are predicted to occur soon, they are promoted to higher (faster) levels of the hierarchy, allowing the CPU to retrieve them more quickly. Nevertheless, caching did not become a prevalent research topic until the 1960s [6].

The memory hierarchy diverges from the RAM model, which assumes a coherent memory where all accesses take the same amount of time. With the hierarchy, this is no longer true. However, the reason the hierarchy managed to embed itself into conventional architectures was the overall system performance gain and the seamless integration into the computer system, where cache management policies run automatically in the background without programmer intervention.

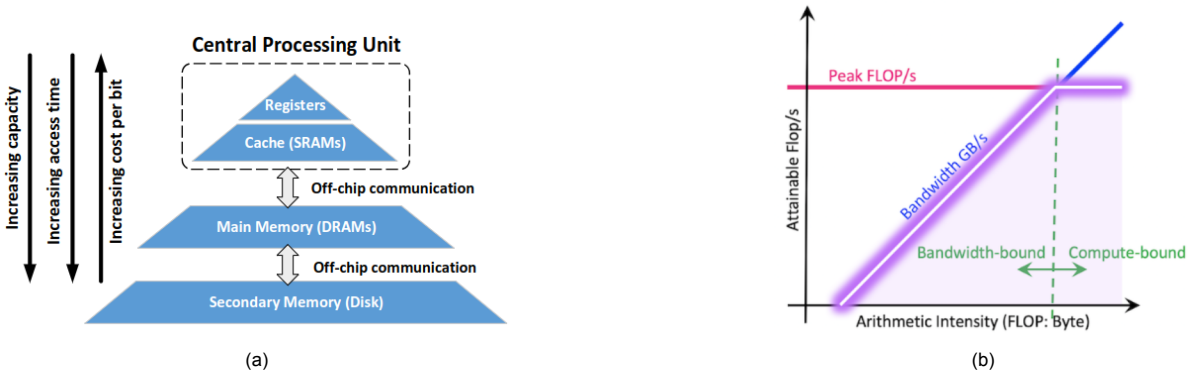


Figure 1.2: (a) The memory hierarchy bridges the gap between the processor and memory, improving data access latency and overall system performance [7, p.3]. (b) Abstract illustration of the attainable performance versus the arithmetic intensity of the processed data. The peak performance for low arithmetic intensity applications is capped by bandwidth limitations [8].

In recent years, applications such as machine learning, graph processing and network monitoring have become popular. These ”data-intensive” applications have low arithmetic intensity meaning that they process a large number of data-points with only a few operations that must be computed per data-point. The memory hierarchy is inefficient for such applications as it does not fully eliminate the consequences of the memory wall. As application data demands keep rising, the data capacity required exceeds the capacity of the higher levels of the memory hierarchy, resulting in frequent costly access to secondary storage. In addition, as shown in Figure 1.2b, another factor that prevents data-intensive applications from reaching their theoretical maximum performance is memory bandwidth that limits data transfer speeds. Consequently, data-intensive applications, will continue to deteriorate in terms of performance and power efficiency as their major bottleneck remains unaddressed and are at risk of becoming practically unsuitable for Von Neumann architectures. In conclusion, the separation of processing and data storage, remains a significant bottleneck despite the use of the memory hierarchy.

1.1.2. Conventional computing timeline

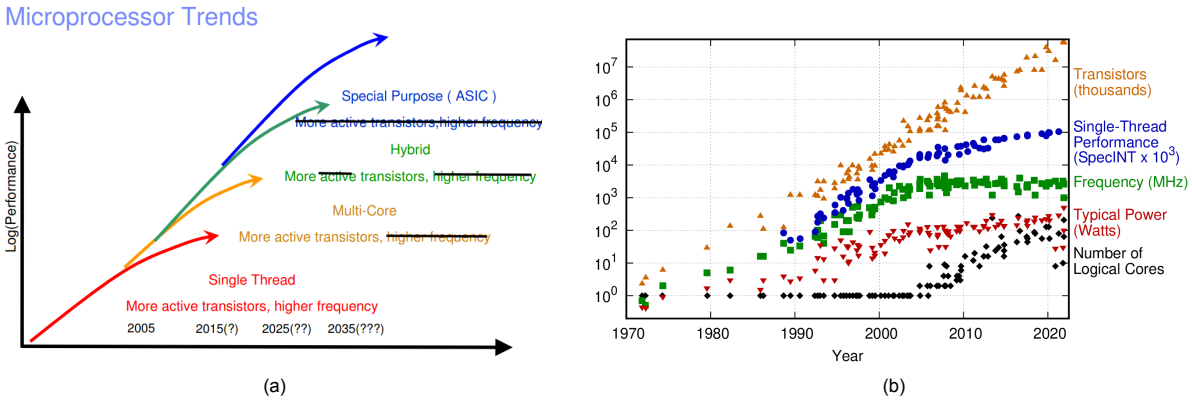


Figure 1.3: (a) Microprocessor trends by P. Hofstee [9]. (b) Evolution of processors over the past 50 years, showing the variation across transistor count, single thread performance, frequency, power and core count, collected by K.Rupp [10].

In Figure 1.3a, P. Hofstee illustrates several partially overlapping periods during which conventional computers evolved to meet the growing demands of applications, despite the constraints imposed by both the Von Neumann model and CMOS technology. Similarly, Figure 1.4 presents a comparable timeline of computing eras, highlighting their respective enablers, constraints, and the evolution of programming models. These transitions are further supported by the data in Figure 1.3b, which reflect the same historical inflection points. This section summarizes the factors that led to each period, culminating in a present where CIM may be the future of computing.

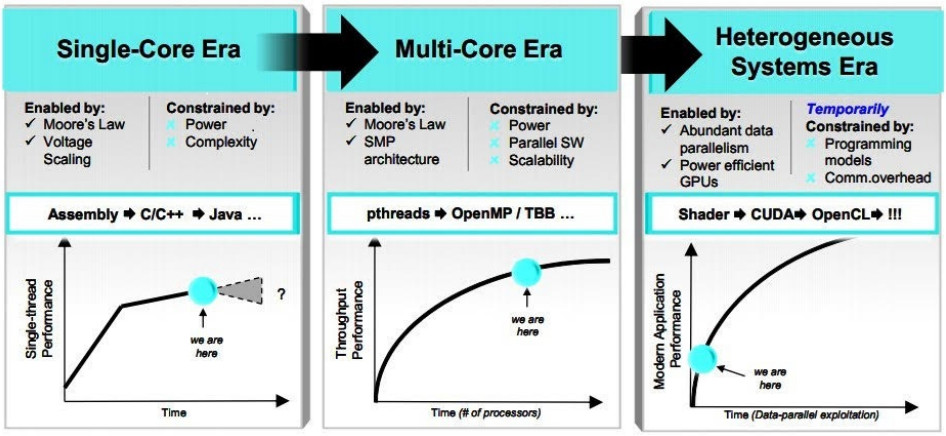


Figure 1.4: Depiction of the various computing eras. ©AMD 2014.

Single-thread era:

Since the 1960s [11], single-threaded computers have met growing application demands largely thanks to transistor downscaling, later formalized by two foundational principles. First, according to Moore's refined law [12], due to transistor downscaling, each biannual chip generation, featured an increased number of active transistors on a chip, enabling higher performance for the same price. Secondly, in accordance with Dennard's (Voltage) Scaling [13], as transistors shrink and are switched at a higher frequency, the power required for the same area remains constant. Together these two principles were the driving force of innovation in the single threaded era.

Concurrently, other research, though less prominent, proved vital for later eras. Research into parallelism began in the 1950s [14, p. 2], leading to supercomputers in the 1960s–1970s, and to the development of multiprocessors and vector processors by the 1980s. Parallelism appeared in many forms and granularities, prompting several classification efforts [15, 16]. Ultimately, two main types of parallelism emerged: data-level, applying the same operation to multiple data elements, and task-level, executing different tasks concurrently. These were leveraged through various techniques, including instruction-level parallelism (ILP), which executes independent instructions simultaneously within a CPU core using pipelining, superscalar, and out-of-order execution.

To abstract parallel and distributed systems for programmers, the Parallel RAM (PRAM) [17] computer model extends the conventional RAM computer model of a computer with a single processing unit. PRAM introduces multiple processing units that can all access a shared global memory in unit time. However, PRAM proved to be an unrealistic model for practical systems for the following two reasons [3, p. 46]. First, it disregards contention issues that arise when multiple processing units access the same memory location simultaneously. Second, the assumption of unit-time access does not hold when the number of processing units increases, making the PRAM model unsuitable for modeling large-scale parallel and distributed systems.

In an attempt to address the above PRAM limitations, the Candidate Type Architecture (CTA) model [18], shown in Figure 1.5, was developed. The CTA model comprises multiple sequential processors connected through an interconnection network, with one processor acting as a controller responsible for initiating computations. Each processor has its own local memory, accessible in unit time, whereas accessing another processor's memory incurs a latency at least an order of magnitude higher. What makes the CTA model particularly general is that it does not impose any specific assumptions on the interconnection topology or the communication mechanism between processors in the system.

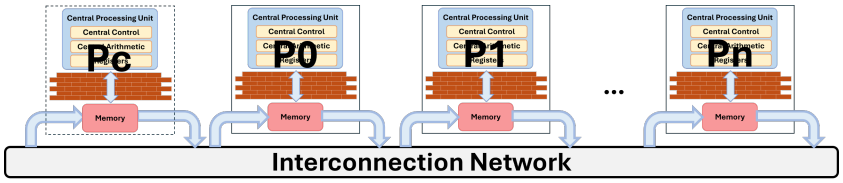


Figure 1.5: The CTA parallel computer model illustrated.

Beyond advances in parallelism, other hardware innovations of the era gave rise to the coprocessors that define modern computing. The concept of the memristor, later key to CIM, was proposed in the 1970s [19]. During the 1970s–1980s, dataflow architectures [20] gained attention, expressing programs as directed acyclic graphs where data moves between instructions. Among these were systolic arrays, in which data streams through a regular network of processing elements that rhythmically exchange partial results. Finally, in 1999, NVIDIA's GeForce 256 was introduced as the first chip marketed as a GPU, a major milestone for real-time graphics acceleration [21].

Programming during this era focused on sequential logic, evolving to improve productivity. Low-level assembly offered fine-grained control but was cumbersome, leading to higher-level languages like C that improved portability. Subsequent languages introduced richer abstractions to manage complexity, while advances in compiler design preserved performance. Functional languages also emerged, though researchers were divided on whether their limited advantages justified introducing a new class of programming languages [22]. Furthermore, the 1990s laid the groundwork for parallel programming, a far more complex task requiring explicit management of communication and synchronization among concurrent tasks. The Message Passing Interface (MPI) [23] became the standard for distributed systems, while for shared-memory machines, Pthreads [24] offered fine-grained synchronization control and OpenMP [25] introduced a higher-level, directive-based model.

Around 2005-2007, Dennard's Scaling was violated as the miniaturization of transistors approached its physical constraints, resulting in stagnating processor frequencies due to excessively high power density. Further transistor reduction resulted in increased leakage current or lower switching speed (i.e frequency). Most notably, the increased leakage current also results in hotter chips that have reached the air cooling limit. This phenomenon is known as the ”power wall” or the ”clock frequency wall” as both frequency and power metrics hit a plateau as shown in Figure 1.3b and thus marking the end of the single-thread era.

Parallelism era:

By the mid-2000s, as frequency scaling reached its limits, hardware designers turned to parallelism, most notably symmetric multicore processor (SMP) scaling, to sustain performance growth. SMPs integrate multiple identical processors on a single chip, all sharing a main memory. This enabled significant performance gains through concurrent execution via ILP and thread-level parallelism. Still driven by Moore's Law, SMP scaling was achieved by adding more cores per chip as transistors continued to shrink, with transistor density at the time expected to double approximately every two years [26].

In addition, much hardware research from the single-thread era resurfaced in an effort to leverage greater parallelism. ILP techniques such as superscalar and out-of-order execution reemerged as key methods for enabling processors to execute multiple instructions in parallel. With the release of CUDA in 2007 [27], general-purpose computing on GPUs (GPGPU) became standardized and broadly accessible, marking the point when GPUs evolved from graphics accelerators into versatile data-parallel computing devices. Both CPUs and GPUs incorporated features inspired by vector processors, including wide data paths and vectorized execution, to increase data-level parallelism. Furthermore, importantly for CIM acceleration, the memristor was physically implemented as a component in 2008 [28]. In parallel, dataflow engines (DFEs) such as those developed by Maxeler Technologies [29] demonstrated the practicality of dataflow-style execution for accelerating highly parallel or irregular workloads. Finally, in 2015, Google's tensor processing units (TPUs) reintroduced systolic array principles to efficiently accelerate machine learning computations [30].

A recurring pattern among emerging technologies is that limited programmability often hinders their adoption, despite potential performance advantages. For instance, although systolic arrays were extensively studied since the 1980s, programming required knowledge of too many low-level implementation details [31], it was not until 2015 that Google's TPU popularized their principles by providing accessible programming interfaces through TensorFlow [32] and later PyTorch [33]. Similarly, NVIDIA's NV1 [34], a predecessor to the GeForce 256, failed due to poor software compatibility and programming complexity. NVIDIA learned from this experience and later introduced CUDA to ensure programmability and broader adoption of GPU computing.

Programming languages also revisited earlier ideas to better express parallelism, focusing primarily on shared-memory systems that aligned with SMP scaling. Pthreads provided a low-level execution model, but over time, higher-level abstractions such as OpenMP, and Intel's Threading Building Blocks (TBBs) [35], which represented computation as task graphs, became more favorable for their ease of

use. Functional programming also regained interest for its parallel-friendly semantics, while compilers increasingly automated parallelism extraction, hardware mapping, and load balancing across cores.

Concurrently, numerous software frameworks were developed that laid the foundation for programming in the hybrid era, with one of the most notable being OpenCL [36, p. 399] in 2009. Originally motivated by the need for an open, GPGPU programming alternative to CUDA, OpenCL has since become an industry standard for heterogeneous computing. Subsequent work, such as OpenACC [37], sought to improve upon OpenCL by making parallelism specifications implicit through compiler directives rather than explicit kernel definitions. Legion [38], on the other hand, adopts a different approach in which the algorithm is separated from the scheduling specification, making optimization and porting easier. Despite their differences, all these frameworks share the common goal of providing scalable, high-level abstractions that enable general purpose hosts to distribute parallel tasks efficiently across multiple connected coprocessors.

Moving closer to the present, two key challenges have hindered further performance scaling through parallelism. First, the “ILP wall” illustrates that as systems approach the theoretical limits of an algorithm’s inherent parallelism, the cost, area, and power required to extract additional ILP increase sharply, resulting in diminishing returns. Second, the “Dark Silicon” phenomenon [39], driven by the end of Dennard’s scaling, arises because although transistor dimensions have continued to shrink, power constraints prevent all transistors from being active simultaneously. As a result, up to 50% of transistors remain inactive at any given time. Consequently, the growth in the number of usable logical cores has stagnated, as shown in Figure 1.3b.

Hybrid era:

Figure 1.3b shows that presently transistors are still downscaling and that single-thread performance is still increasing. While the latter is attributed due to innovations in compilers, the former is expected to end once the ”Physics wall” [40] is reached. Thus, to leverage more performance we are transitioning to heterogeneous computing, where systems leverage multiple, specialized coprocessors to handle different types of workloads more efficiently. Examples include offloading massively data-parallel programs to GPGPUs, deep pipelines to DFEs, machine learning workloads to TPUs and adaptive applications to reconfigurable computers based on field programmable gate arrays (FPGAs).

One of the challenges we now face is the emergence of the ”AI memory wall” [41]. The rapid adoption of AI technologies has driven an exponential increase in computational demand, met largely by advances in GPUs and TPUs. However, as shown in Figure 1.6, while peak FLOPS have scaled aggressively, memory bandwidth has lagged behind significantly. This growing imbalance threatens to make future AI workloads increasingly unsustainable, as the memory bottleneck becomes the dominant limiter of performance. Hence, we have come full circle—returning to the memory wall. To address this, researchers are now investigating CIM accelerators, a class of coprocessors designed specifically to alleviate the memory bandwidth crisis by performing computation directly where the data is stored.

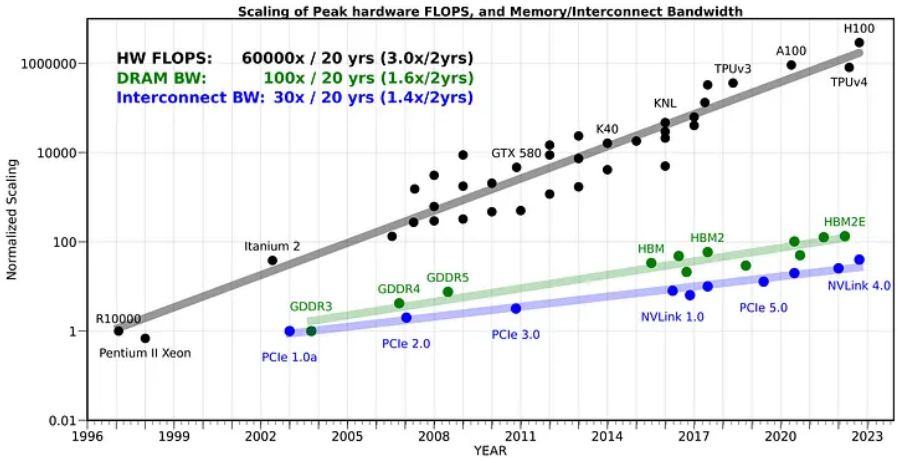


Figure 1.6: The scaling of the bandwidth of different generations of interconnections and memory, as well as the Peak FLOPS. As can be seen, the bandwidth is increasing very slowly [41, Fig. 1]

1.1.3. CIM accelerators

Computing-in-memory (CIM) accelerators [7] are a class of promising emerging coprocessors. The idea is to augment conventional computers with a unit that can perform memory storage and computation in place. This can be achieved by a combination of operations that are performed either directly in the CIM unit's memory array or in the array's digital periphery. The most popular accelerated operation is by far vector-matrix multiplications (VMMs) which can be executed in one shot, but others include boolean operations and key-value store operations directly in the memory array, eliminating the need to transfer data to and from the CPU. Consequently, CIM is especially beneficial for data-intensive, memory-bound, highly parallel workloads where the cost of moving data dominates both performance and energy consumption. These workloads, which include convolutional neural network (CNN) inference, regular expression matching, partial differential equation (PDE) solvers, and integer sorting, are collectively referred to as CIM applications. As these workloads continue to grow in size and complexity, CIM offers a scalable path forward by addressing the fundamental limitations of memory bandwidth and energy efficiency that conventional Von Neumann architectures struggle to overcome.

CIM accelerator research is still in its infancy, while numerous application specific designs with varying levels of complexity have been proposed. Ongoing research [7] is focused on improving hardware such as minimizing device non-idealities, designing both, primitive and complex operations at the circuit level, and defining effective architectural models. CIM's departure from the Von Neumann model, while its defining strength, also constitutes its greatest challenge in software. Specifically, the Von Neumann architecture, shown in Figure 1.7a, maintains a clear separation between computation, memory, and control, which simplifies program reasoning but also implicitly leads to the memory wall. In contrast, CIM is a hierarchy of micro-units that each contain computation, memory and dedicated control functions. The alleviation of the memory wall comes at the cost of increased complexity, as illustrated in Figure 1.7b. This Venn diagram depicts the now overlapping concerns—memory arrays that also perform computation and localized control placed near the memory to manage operations.

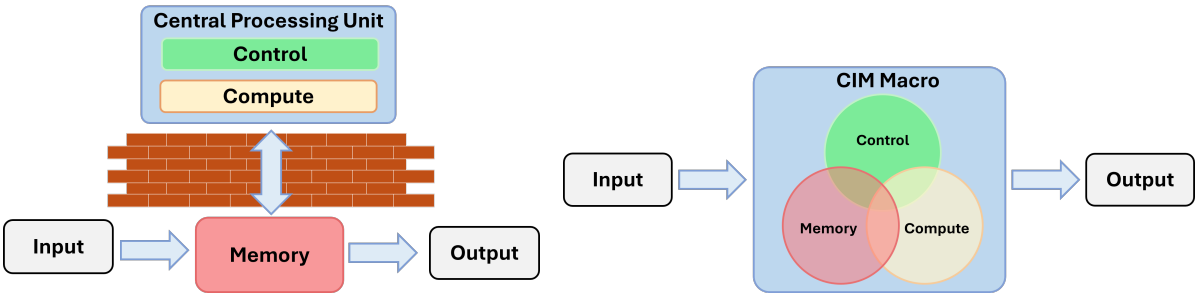


Figure 1.7: (a) The Von Neumann model compared to (b) CIM macro expressed using Von Neumann model's components

As CIM accelerators are typically a hierarchy of interconnected CIM components that collectively perform the computation independent of the specific interconnect and organizations across design instances, an analogy is frequently drawn between CIM and conventional distributed systems. They both require careful handling of concurrency, data distribution, and inter-component communication, making their programming challenging [42, p. 3]. As illustrated in Figure 1.8, the CTA model can be effectively adapted to multi-CIM-macro systems, capturing non-trivial interactions between modules by realistically representing macro-to-macro latency costs.

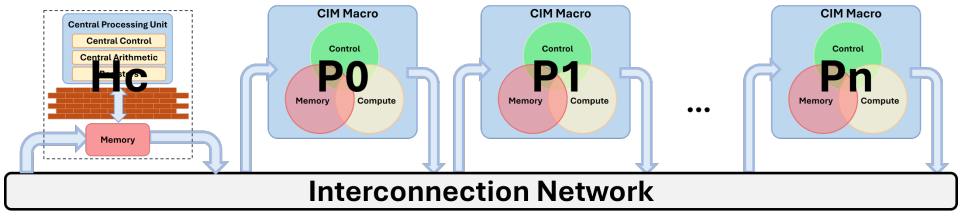


Figure 1.8: CIM visualized using the CTA, with a general purpose host controller and multiple interconnected CIM macros.

In programming, there are two major contrasting paradigms. Imperative, or conventional, programming specifies how to perform a task step by step, whereas declarative programming specifies what should be done, leaving the details of execution to the compiler or runtime. Backus [43] argued that conventional programming languages mirror the architectural limitations of the von Neumann model, as they force programs to represent computation as sequential updates to a shared memory state resulting in a bloated language with limited expressiveness. In contrast, in the context of CIM, most prior work suggests that, given the high complexity and variability across implementations, a purely declarative, API-based approach is desirable to abstract away hardware details. However, such an approach also limits expressiveness, as programmers are constrained by rigid APIs and have little control over optimization decisions. A well-designed CIM programming model should therefore strike a balance by empowering programmers to guide optimizations through high-level functional abstractions while relying on compilers to manage the low-level, implementation-specific details that are often too complex to handle manually.

As a result, a CIM programming model is seen as essential not only because it abstracts hardware complexity, but also because it facilitates efficient performance optimizations. For CIM accelerators to become practically useful coprocessors to conventional architectures, applications must be carefully mapped to exploit their unique strengths. Without the right abstractions and scheduling control, programmers risk under-utilizing the hardware and losing the efficiency benefits that make CIM attractive in the first place. In addition, we should avoid repeating the trajectory of other technologies with great potential but poor programmability, with systolic arrays described in Section 1.1.2 as the most prominent example. Despite the initial interest driven by their potential performance benefits for regular computations, their programming complexity kept them from mainstream use for nearly forty years. Only recently they have been rediscovered, with the TPUs and the tensor math-based frameworks that popularized them. Instead, we should take inspiration from GPGPUs and the introduction of CUDA, which abstracted away hardware complexity and provided a unified programming model that made GPU computing broadly accessible to developers beyond only computer graphics specialists. Therefore, an ideal programming model for CIM should provide intuitive, general abstractions that apply across a wide range of application domains, enabling application developers to write programs without requiring low-level hardware knowledge.

1.2. CIM programming model challenges

Although hardware remains the primary roadblock to widespread CIM adoption, in this work we choose to look ahead by exploring the software foundations that will shape future CIM-based systems. Thus, we focus on the programming models that will provide abstractions to help manage the hardware complexity, enabling developers to express computations in an intuitive way while striking a balance between performance, portability, expressiveness and ease of use. Nevertheless, looking this far ahead, while interesting, brings forth its own set of challenges:

- **Few programmable accelerators:** Currently, CIM accelerators are most commonly fixed-function ASICs because their architectures are designed around accelerating a very specific workload. To use such a device, one simply provides values, after which the desired outputs are produced. Some of these devices also allow the user to specify certain configuration parameters. Hence, there is no need for programming because the device is inflexible, limited to executing the specific functions it was designed for. Programmable CIM accelerators do exist, but they have received comparatively less research attention, as the increased flexibility comes at the expense of increased complexity and often a decrease in performance. This limited focus in research means there is less prior work to build upon and potentially many issues that remain undiscovered.

- **Multiple microarchitectures exist**: A variety of microarchitectural designs exist that are application-specific, meaning each accelerator specializes in a limited set of applications. Consequently, CIM accelerators can differ significantly in the underlying technology used for computation, the hierarchy of components employed, and the granularity available for programmers to interact with the device. Currently, for programmers to utilize one of these accelerators, extensive knowledge of its design is necessary to achieve optimal performance, akin to the detailed understanding required for assembly programming in conventional computers. Uniting all of these vastly different implementations under a single programming model may prove to be impossible.

- **Variations across applications**: Several CIM applications operate in diverse scientific domains. According to M. Z. Zahedi [7, p. 33], for a programming model to generalize across such varied use cases, it must support varying data types and sizes, different operations and access patterns. In addition depending on the hardware such a programming model would need to support different accuracy reductions, diverse communication patterns between micro and macro units, and flexible ways of communication and synchronization with the host.

- **Exploiting Parallelism:** In order for CIM to be useful it must be utilized effectively to exploit as much parallelism as possible from applications to maximize performance. Leveraging parallelism effectively is notoriously difficult, even in conventional computing systems, and becomes even more challenging in the context of CIM. This is because CIM accelerators have a lot of degrees of freedom when it comes to parallelism as both micro and macro units in the accelerator's structure are able to exploit both data and task level parallelism. Thus, programs should be written in a manner that makes parallelism both easy to express and extract.

- **Shortcomings of Prior Work**: It is important to note that relevant work to CIM programming does exists [44, 45, 46, 47]. However, all of these examples exhibit at least one of the three shortcomings. First, most are too low-level, device-specific and hence lack portability: programs written for one accelerator cannot be reused on another, forcing programmers to repeatedly rewrite applications when switching to a new architecture. Second, several are application-specific falling short in expressiveness: they may support a narrow class of applications, such as neural networks, while neglecting others with different characteristics, such as search operations. Third, these tools are purely declarative. As a result, developers have little to no control over critical aspects such as parallelization strategies, data movement, or memory hierarchy utilization. All optimization decisions are deferred to the compiler, which may struggle to make optimal choices.

1.3. Research topics

The previous sections have outlined how computing has progressed toward CIM and why developing a dedicated programming model is important. We then discussed the numerous challenges that programming models must eventually face. We define an ideal programming model as the one that addresses all of the challenges outlined in Section 1.2 and, as a result, is capable of unifying both diverse CIM applications and different hardware implementations. At the same time, it should deliver high performance with low overhead on any underlying hardware while providing an intuitive and expressive programming experience. However, CIM devices, circuits, and architectures are still in their infancy, making it too early to validate a unified programming model across hardware platforms. Such validation would require access to multiple programmable accelerators, which are currently scarce, whether simulated or otherwise. In addition, developing a compiler for these accelerators would be necessary, however this is not feasible within the time limitations of this thesis.

Therefore, in this work, we focus on proposing opportunities and identifying limitations for future CIM programming models. We achieve this goal by analyzing existing CIM software and popular programming models to derive a set of essential meta-abstractions that an ideal programming model should integrate—either by incorporating effective ideas from related CIM frameworks, filling gaps where current approaches fall short, or aligning with programming models from more mature domains to ensure long-term success and sustainability. To consolidate these suggestions into a coherent foundation, we design a general purpose programming model, which we define as one that unifies a broad range of CIM applications under a common set of abstractions. We then validate this design by expressing a variety of workloads from different domains using the proposed abstractions. In addition, based on our findings, we suggest how future CIM hardware research should be directed to align more closely with the needs of software development. It is important to note, however, that while our models provide guidelines for hardware design and program-to-hardware mapping, neither can be fully verified due to the current immaturity of the underlying technology.

In short, this thesis aims to uncover the answer to the following research question:

Research Question (RQ): Is it possible to design a programming model for CIM accelerators that offers abstractions capable of expressing applications from diverse domains that can benefit from CIM acceleration?

In the following, we divide the above research question into two subquestions. Then, we elaborate

on how we aim to answer each of them. These subquestions will also serve as reference points when presenting and discussing the results in later sections.

Subquestion 1 (SQ1): What essential meta-abstractions should CIM programming models possess?

Programming abstractions are always grounded in underlying concepts they aim to express or promote to programmers. In this work, we ask which concepts CIM programming models should implement through their abstractions. Although different abstractions may realize the same concept in various ways, identifying the essential meta-abstractions can serve as a blueprint for designing or adopting new abstractions for CIM. This allows developers to evaluate an abstraction by asking: what capability does this abstraction provide?

To answer this question, we first require some prerequisite understanding of CIM accelerators. This includes identifying the commonalities and challenges that arise across different architectures, as well as determining which aspects are relevant and useful to programmers and which should be abstracted away. Subsequently, we analyze both CIM-specific and conventional programming models to examine which meta-abstractions existing work prioritizes. This subquestion is addressed by compiling a comprehensive list of desirable meta-abstractions for CIM programming models in Section 3.3.

Subquestion 2 (SQ2): Do there exist abstractions that facilitate applications from different domains in a CIM programming model?

Once we determine what meta-abstractions should be expressed in **SQ1**, the next goal is to identify effective abstractions that convey this meaning. Our methodology is as follows: first, as prerequisite knowledge, we examine three CIM applications that benefit from CIM acceleration and their corresponding implementations. Subsequently, we construct a programming model capable of expressing CIM applications from different domains, using the results from **SQ1**. Finally, we evaluate the model's expressiveness by implementing these applications within it.

1.4. Thesis contributions

This thesis presents four main contributions. First, in Section 3.3, to answer **SQ1**, we define a set of desirable meta-abstractions for CIM programming models: implicit scheduling, hardware generalization, optimization, and explicit expression of parallelism. These meta-abstractions were identified by examining abstractions used in both CIM-oriented systems and other computing domains, such as heterogeneous, distributed, and dataflow systems. We also discuss which specific abstractions could instantiate each meta-abstraction within a new CIM programming model.

Second, in Section 4.1, we present a possible solution by proposing a concrete programming model for CIM inspired by Halide [48]. This model is realized through the selected meta-abstractions (as defined in the first contribution) and is implemented using a set of general abstractions which we thoroughly explain. To briefly summarize, these abstractions are: a deferred execution model, a general platform model, separation of algorithm from scheduling, shared variable types and declarative scheduling directives. We later provide concrete syntax and semantics for these elements to form a usable Domain-Specific Language (DSL). This work thereby demonstrates how the abstract recommendations can be materialized into a practical design.

Third, in Section 4.2, we evaluate the expressiveness of the proposed CIM programming model across diverse application spaces to answer **SQ2**. Specifically, we implement three practical applications: integer sorting, a Database workload; network-monitoring pattern matching, a security application; and convolutional neural network inference, a machine learning application.

Lastly, in Chapter 5, we provide insights into the development process of this thesis to guide future research. This discussion covers various pathways for extending the proposed programming model or exploring alternative approaches not yet fully investigated. Furthermore, we outline key pitfalls encountered during development to caution and inform subsequent work. This comprehensive analysis serves to lay the foundation for future programming models for CIM.

1.5. Thesis structure

This thesis is structured as follows.

Chapter 2 elaborates on the concepts of CIM accelerators and programming models. First, it summarizes the fundamental concepts of CIM accelerators in a hierarchical bottom-up manner. Second, it presents three concrete CIM application implementations developed for the verification step of our programming model. Finally, the chapter concludes with an in-depth discussion of programming models and related research which we analyze in later chapters.

Chapter 3 presents an analysis of existing abstractions, both conventional and CIM-specific, with the goal of identifying essential meta-abstractions that can guide the development of future CIM programming models. Based on the identified meta-abstractions, it proposes a set of concrete abstractions that can be used to realize them within a programming model.

Chapter 4 provides the detailed specifications of the proposed CIM programming model based on the results of Chapter 3. Subsequently, the model is utilized to reproduce the three CIM applications discussed in the Chapter 2.

Chapter 5 discusses the challenges encountered throughout the course of this project and outline potential directions for future work. We also reflect on lessons learned during the development of our proposed programming model and how these insights can inform subsequent research.

Chapter 6 concludes the thesis and reflects on the contribution.

2

Background

This chapter provides background information on both Computing-in-Memory (CIM) accelerators and programming models that influenced our proposed CIM programming model. First, Section 2.1, summarizes fundamental concepts underlying CIM accelerators across multiple abstraction layers. Second, Section 2.2, presents three practical CIM application implementations for the verification step of our proposed programming model. Finally, Section 2.3 offers an in-depth discussion on programming models and related research which we analyze in later chapters.

2.1. Fundamentals of CIM accelerators

In this section, we discuss fundamental concepts of CIM accelerators. We begin with a brief overview of the hardware across different abstraction levels, following a bottom-up approach. These levels include memory cell technology, circuit design and system architecture. The goal is to provide readers with the necessary intuition for understanding the inner workings of CIM accelerators and to understand certain design decisions behind our proposed programming model.

2.1.1. Memory cell technology

There are many types of memory cells that can be used to implement CIM. However, to limit the scope of this work, we focus on the memristive technology family, as its properties have been most widely leveraged for vector-matrix multiplication (VMM), the predominant accelerated operation in the domain. The memristor [19], illustrated in Figure 2.1, is an abstract electronic component that encompasses all memristive memory cells. Various technologies—such as ReRAM, PCM, MRAM and FeFET [7, p.17]—can be used to construct a memristor, but at a high level it functions as an analog memory element that stores data by altering its conductance. While a memristor is theoretically capable of higher than binary resolutions, physical constraints such as noise and state degradation limit most practical implementation to two states: the high resistive state (HRS) and the low resistive state (LRS).

We show a comparison of the characteristics of each emerging memristive cell type and mainstream memory type in Table 2.1. All emerging memristive technologies are non-volatile, meaning they retain their internal state even after power is removed. Furthermore, their feature sizes are comparable to DRAM and FLASH memory. In terms of performance, they have high latency similar to DRAM and FLASH. However, their most significant limitations is that they require up to several orders of magnitude more reprogramming energy per bit than SRAM and DRAM, while offering several orders of magnitude lower endurance than both. The former undermines the very energy efficiency benefits that CIM is intended to provide, while the latter causes memristive devices to gradually degrade with repeated overwrites. As noted by Zahedi [7, p.19], these two limitations are the primary reasons why memristive devices are used only in applications that require infrequent reprogramming and are thus unsuitable for implementing cache or main memory structures.

From this analysis, we draw several conclusions relevant to the design of our CIM programming model. First, to account for the high reprogramming energy and limited endurance of memristive devices, we made the decision to incorporate immutability constraints into the programming model. By explicitly discouraging frequent memory rewrites, the programming model not only aids in extending

device longevity but also discourages programmers from structuring energy inefficient applications. Second, given that memristive technologies are not integrated into main memory or cache hierarchies, we treat CIM units as specialized coprocessors. This motivates a heterogeneous programming model where accelerators are explicitly managed alongside the host processor.

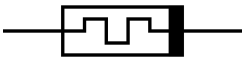


Figure 2.1: The electronic symbol for the memristor.

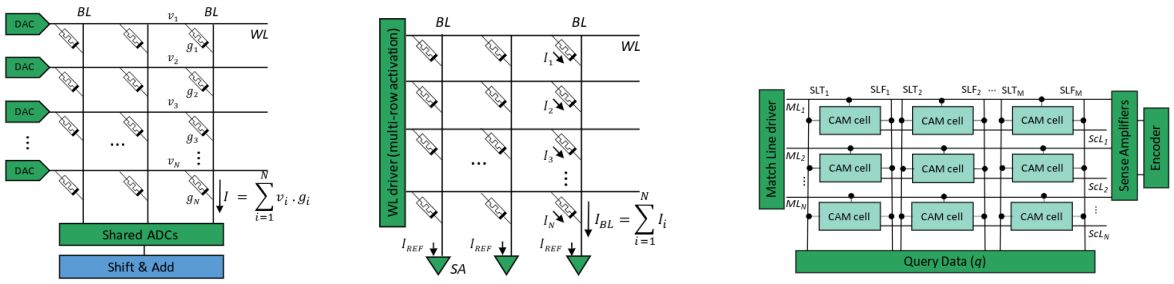
	Mainstream			Emerging			
Device	**SRAM**	**DRAM**	**FLASH**	**RRAM**	**PCM**	**STT-MRAM**	**FeFET**
Write latency	~1ns	~10ns	0.1–1ms	~10ns	~10ns	>5ns	~10ns
Read latency	~1ns	~3ns	~10ns	~10ns	~10ns	>5ns	~10ns
Non-Volatile	No	No	Yes	Yes	Yes	Yes	Yes
Write energy (per bit)	~1fJ	~10fJ	~100pJ	~1pJ	~10pJ	~1pJ	~ 0.1pJ
Density (F^2)	$120-150$	$10-30$	$10-30$	$10-30$	$10-30$	$10-30$	$10-30$
Endurance	10^{16}	10^{16}	10^4-10^5	$> 10^7$	$> 10^{12}$	10^{15}	$> 10^5$

Table 2.1: Comparison of mainstream and emerging memory technologies [40]. F denotes the feature size of the technology.

2.1.2. CIM circuit primitives

A memristor on its own functions as a storage element, but it can also be used to implement in-place computations. By in-place computations, we mean operations that require little to no data movement compared to conventional computers. Depending on the type of in-place operation and its implementation, computation is performed either directly within the memory array or in close proximity using peripheral circuitry.

A CIM primitive is one of the three types of scalable 2D memristor array structures that Reis et al. [1] have classified, shown in Figure 2.2. Each of the CIM primitives is capable of performing a different set of operations. In addition, depending on the target application and the required precision, the dimensions of the memory arrays might differ. We briefly discuss each CIM primitive below:



(a) CIM-Xbar for VMM.

(b) CIM-ALU for boolean logic.

(c) CIM CAM structure.

Figure 2.2: CIM primitives [1] illustrated by Khan et al. [49].

- **CIM Crossbar (CIM-Xbar):** The most popular CIM primitive is the crossbar, as it can perform VMM in constant time and has been widely used in machine learning. In this structure, memristors are placed at the intersections of bitlines (BLs) and wordlines (WLs), with the matrix elements encoded as conductance values and the input vector applied as voltages along the wordlines. By Kirchhoff's laws, the resulting currents are produced at the ends of the bitlines, where they are sensed to obtain the output. Figure 2.2a shows an example calculation for a single BL.

  While most CIM-Xbar implementations physically perform VMM, some instead apply the input vector along the bitlines to realize matrix–vector multiplication (MVM) using the same crossbar

structure [50]. For a programming model, this design choice does not matter, as VMM and MVM can be converted into one another through matrix transposition, a transformation that can be handled by the compiler. We make this distinction explicit because in our examples we conceptualize operations as MVMs for clarity but illustrate them as VMMs on hardware, since this is the dominant form used in the field.

- **CIM arithmetic and boolean logic unit (CIM-ALU):** Same structure as the CIM-Xbar but used differently. Specifically, operands are stored in different WLs while each BL is a bit position. This allows multiple WLs to activate simultaneously and act as a vector processor, applying either arithmetic or boolean operations at each operand in parallel. Many operations are possible that all have varying access patterns, requirements and implementations. On its own, this is the least-used primitive, as depending on the implementation it can require many memory cell rewrites, leading to higher energy consumption and reduced device lifetime. Due to its structural similarity to the CIM-Xbar, several implementations combine features of both [51].

- **CIM Content Addressable Memory (CIM-CAM):** Differs the most out of all CIM primitives as memristor cells are used to create more complicated CAM cells. Subsequently, look up tables are distributed and encoded across multiple memory arrays. Performs key-value store operations which include massively parallel matching, comparison and distance computations.

Ideally, our programming model should support all three CIM primitives and their respective applications. That is why we originally aimed to implement applications for each CIM primitive to evaluate the expressiveness of our proposed programming model. However, as discussed in more detail in Section 5.2, implementing a CIM-ALU application proved challenging, as we found no CIM accelerators that support it in practice for implementing real applications.

Most accelerators are homogeneous, meaning they implement only a single type of CIM primitive with uniform specifications. A few exceptions exist, such as architectures that combine the functionality of both CIM-ALU and CIM-Xbar structures [51], leveraging their structural similarity. Regardless, this implies that a potential compiler must be aware of the specific primitive operations supported by the target accelerator and raise an error when unsupported operations are used.

2.1.3. Memory array peripheral

A complete CIM processing unit or CIM macro is composed of a CIM primitive memory array accompanied by peripheral circuitry. In Figure 2.3 Zahedi [7] provides an abstract view of a CIM macro containing a CIM-Xbar. The CIM macro is composed of four circuits that can be pipelined: input processing, the CIM-Xbar, sensing, and output processing. Input processing receives data from an external source and performs the necessary pre-processing—such as buffering and digital-to-analog conversion (DAC)—before feeding the signals into the CIM-Xbar. The CIM-Xbar array then performs the computation. Next, a sensing circuit captures the CIM-Xbar's outputs and performs analog-to-digital conversion (ADC). Finally, output processing applies any required post-processing to the CIM-Xbar's results. To facilitate synchronization across these pipeline stages, a controller is employed to execute instructions and coordinate their operation.

In an effort to generalize and conceptualize Figure 2.3 for all CIM primitives from a programming perspective, we created the illustrations shown in Figure 2.4. Figure 2.4a presents the abstract view of a CIM macro from Section 1.1.3, which we then refine and expand into a more detailed representation in Figure 2.4b. In this model, the CIM macro contains a CIM primitive that serves both as memory and as a compute unit. Additionally, it includes two dedicated compute units—one for pre-processing and one for post-processing. Buffers are placed at the inputs and outputs to support pipelining, and we distinguish between local control, specific to each CIM macro, and global control, which coordinates the entire CIM accelerator. This is a very crude view of a CIM macro as not all instances have every component listed such as the buffers but it offers a clearer view of the CIM macro to programmers.

The periphery of a CIM primitive introduces significant complexity in both design and programming, as it varies depending on the target application. Post-processing, for example, may range from simple adders [52] to full-fledged functional units [45, 46] attached to each CIM primitive. Consequently, the set of complex operations a CIM primitive can execute with the aid of its periphery is not fixed, requiring compilers to verify whether a given operation is supported.

Another challenge related to the periphery is balancing the circuit stages to form an efficient pipeline [7, Sec 6.3]. Different operations may require varying durations in each pipeline stage, depending on the setup and post-processing involved. Even the same operation may exhibit different pipeline latencies depending on its scale, primarily due to the analog nature of the memristor and the use of DAC and ADC converters. For example, for the CIM-Xbar, multiple bitlines often share a single ADC, as providing one per wordline is too costly and impractical. As a result, bitlines sharing an ADC must be sensed sequentially. Although such details are low-level, programmers or compiler developers aiming to fully exploit the accelerator's performance must account for these variations during development.

Ultimately, for the our proposed programming model we decided that manipulating the innerworkings of a CIM macro is too low-level. We aimed for a higher level programming model that values expressiveness over fine-grain control. However, it could potentially be expanded to add finer-grained control of the CIM macro with future work as discussed in Section 5.3.

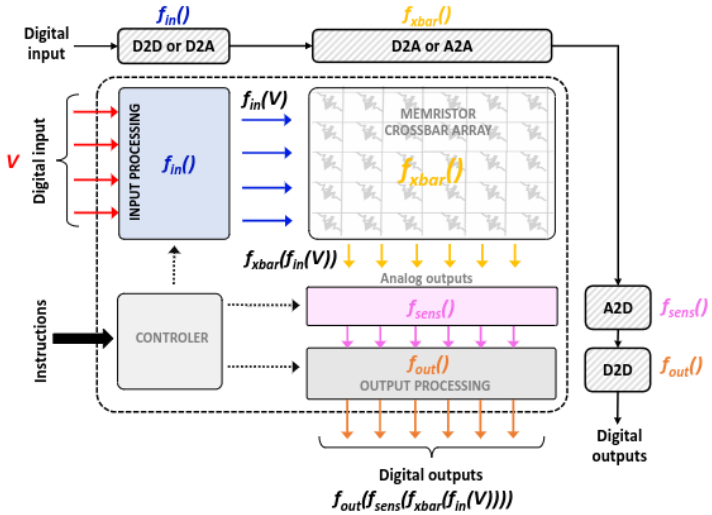


Figure 2.3: Illustration of the periphery of a CIM macro by M. Z. Zahedi [7, p. 25] comprising of four parts: 1) input processing, 2) a CIM-Xbar, 3) sensing, 4) output processing

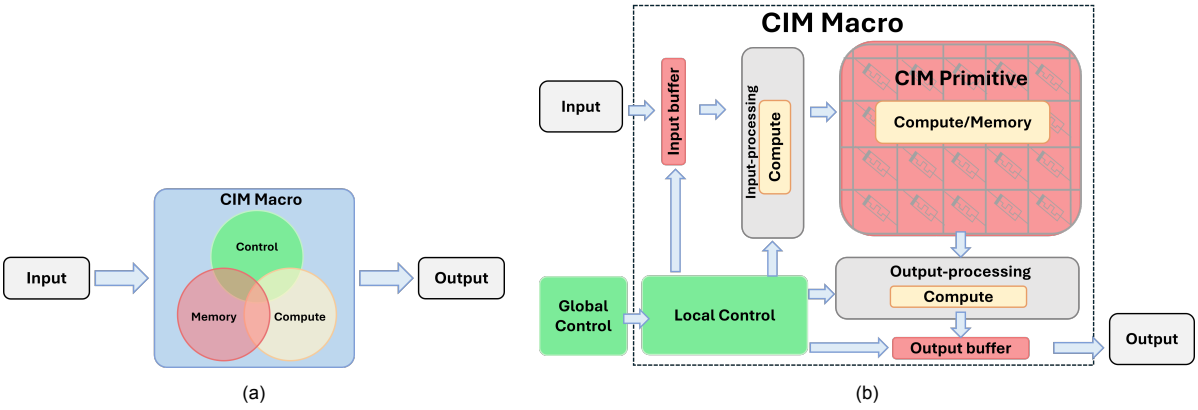


Figure 2.4: (a)Abstract view of a CIM macro. (b) More detailed illustration of a CIM macro from the programmers perspective.

2.1.4. System architecture

One of the key architectural decisions in CIM systems is the choice of hardware organization. As shown in Figure 2.5, Sun et al. [53] identify four envisioned models: pipelined, homogeneous, heterogeneous, and distributed. Pipelined designs (Figure 2.5a) enable cascaded primitive operations across memory arrays with minimal periphery, forming deep computation pipelines. Homogeneous designs (Figure 2.5b) treat identical arrays as cores that share peripheral circuitry, allowing more complex op-

erations through coordination. Heterogeneous designs (Figure 2.5c) incorporate additional processing units to support operations that are inefficient or impractical in CIM. Finally, distributed designs (Figure 2.5d) divide computation across multiple CIM units that communicate, similar to distributed computing systems. This classification is not exhaustive, as many existing architectures are hybrids that combine elements from multiple models. While a CIM accelerator typically implies a heterogeneous organization at the system level, its internal organization can follow any of the models described above.

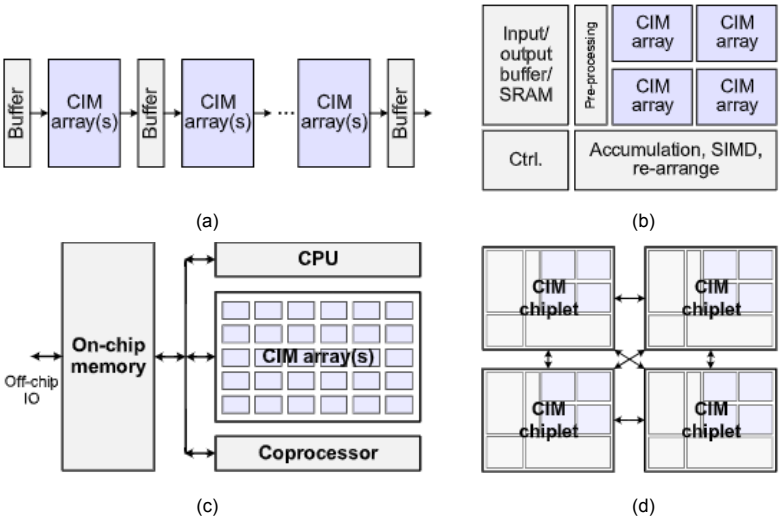


Figure 2.5: Several CIM organizations illustrated by Sun et al. [53], each inspired by a conventional architectural paradigm: (a) Pipelined, (b) Homogeneous, (c) Heterogeneous, (d) Distributed.

An ideal programming model should be able to unify different hardware organizations. However, this generalization makes abstraction design more challenging, as the abstractions must be broad enough to target all supported organizations. At the same time, the programming model must express performance-critical behavior that depends on the specific organization. This can be addressed by exposing organization-specific directives for optimization, or by hiding the complexity behind the compiler—though the latter may come at the cost of reduced performance. In addition, the cost of operations—particularly data movement—can vary significantly across organizations, requiring either explicit scheduling mechanisms or a sophisticated scheduler to manage execution efficiently.

Another key challenge for CIM programming models is the lack of Instruction Set Architectures (ISAs). According to Drebes et al. [47], most accelerators adopt an API rather than an ISA because of the high cost of developing and maintaining an ISA, as well as the reduction in memory array density caused by the inclusion of hardware decoders. Without an ISA, there is no fixed instruction set for the programming model to target. As a result, more responsibility shifts to the compiler and the programming model, since programs must be lowered to hardware-specific APIs. This increases the complexity of the mapping process, as discussed in Section 5.5, since supporting a new architecture requires modifying or extending the compiler.

2.2. CIM applications

CIM is especially beneficial for data-intensive, memory-bound, highly parallel workloads as explained in Section 1.1.3. In this section we present three application implementations, shown in Table 2.2. Our goal is to use these examples when verifying the expressiveness of our programming model. We do this by recreating each application as if we had access to a programmable accelerator compatible with its implementation.

2.2.1. Integer sorting

We selected integer sorting because it represents a comparatively simple and well-understood workload. In this section, we examine sorting in CIM as defined by Liu et al. [54] through their MemSort accelerator. MemSort is a non-programmable accelerator that demonstrates how CIM can be exploited to implement both counting sort and merge sort with linear time complexity. For the purposes of this

Application	Domain	Reference Paper	CIM primitive
Integer Sorting	Databases	Memsort [54]	CIM-CAM
Pattern Matching	Security	CAMASIM [55]	CIM-CAM
Convolutional Neural Network Inference	Machine Learning	ISAAC [52], CIM Explorer [56]	CIM-Xbar/ALU

Table 2.2: List of all CIM application analyzed in this section. For each application we note the paper that describes the implementation, and the CIM primitive it uses.

work, however, we focus only on counting sort, as a single implementation is sufficient to highlight the relevant concepts without overemphasizing sorting itself.

MemSort's memory arrays, that are based on the CIM-CAM primitive are shown in Figure 2.6a. They function as comparison units of size 64x64 bits capable of intra-array sorting. Each row stores a 64-bit integer, while each column corresponds to a bit position. Each element in the memory array is sequentially fed to the input buffer where a parallel comparisons between all stored words and the input buffer is triggered. The comparison results indicate which elements are strictly smaller than the input word, and these results are forwarded to a popcount post-processing circuit that sums the values. The resulting count specifies the index at which the input word should be placed in the sorted array. Finally, a copy of the input is placed at a sorted array buffer as shown in Figure 2.6b. This process is repeated for all words stored in the memory array. In addition, memory arrays are able to communicate with each other using an I/O Buffer.

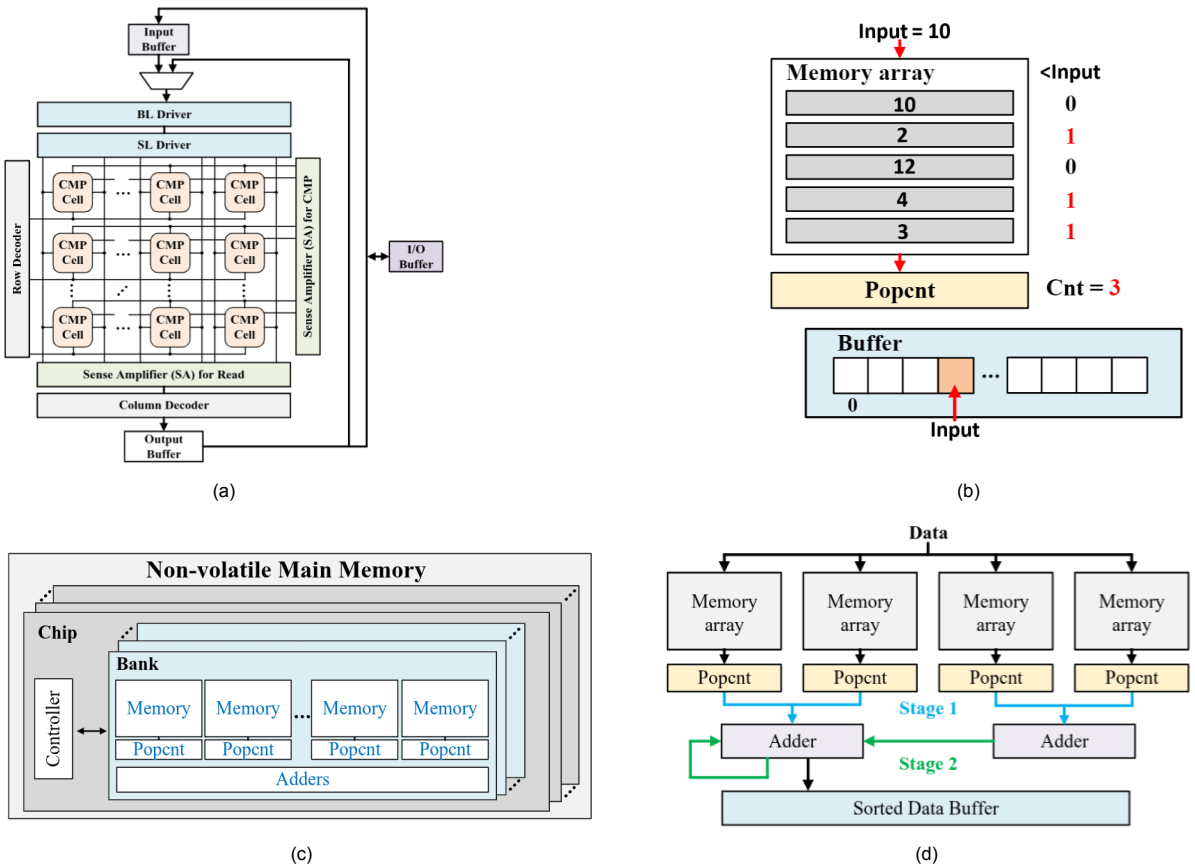


Figure 2.6: (a) Memsort comparison CIM macro, (b) Memsort intra-array example, (c) Memsort hierarchy, (d) Inter-array sorting example using four memory arrays [54].

Due to the limited size of memory arrays, an array–bank–chip hierarchy is introduced, as shown in Figure 2.6c. Each memory array has a dedicated popcount unit, while an adder is shared between every two arrays. Inter-array sorting is implemented across multiple arrays in a manner similar to intra-array sorting as shown in Figure 2.6d. First, a single word is broadcast to the input buffers of all memory arrays. Each array then computes its local popcount result, after which the results are aggregated

through adders until a single value remains, representing the index of the input element in the sorted array. We note that since the accelerator is non-programmable, processing proceeds in lockstep.

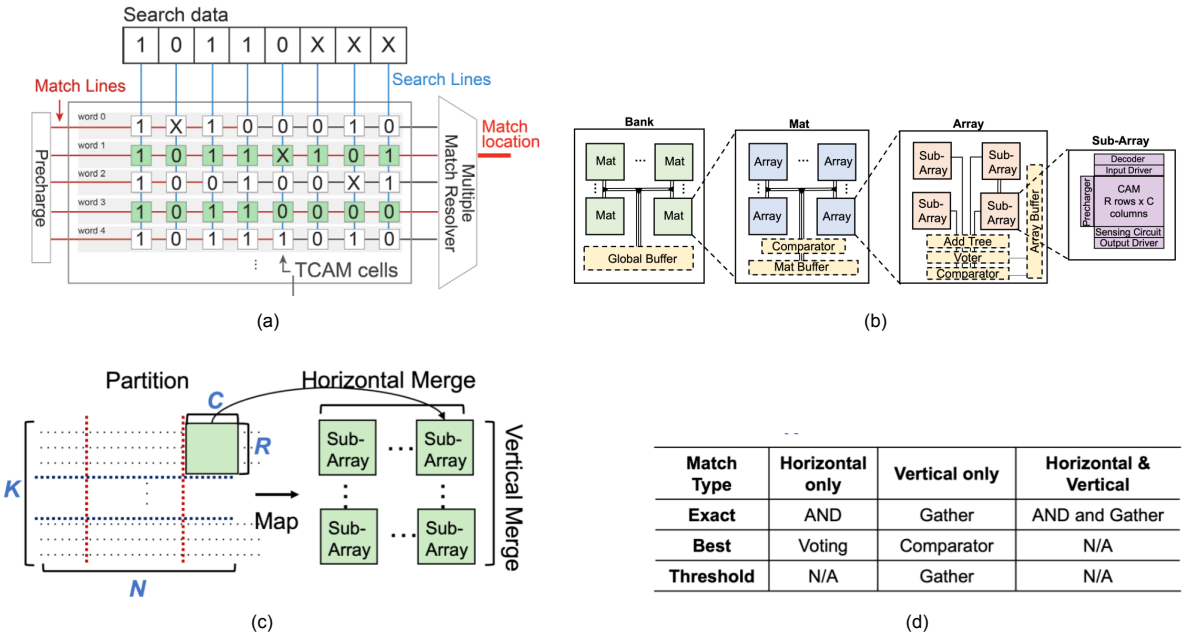
This implementation of sorting is highly compatible with CIM. Notably, during a single sort operation the memory array cells are never overwritten, thereby preserving the endurance of the device. In terms of performance, each number is compared with all other numbers in parallel, yielding constant-time comparison. The aggregation across memory arrays for K adders which are strictly less than the number of elements occurs in $O(\log_2 K)$ time. Repeating this process for all elements to be sorted results in an overall time complexity of $O(n \log_2 K) \approx O(n)$.

There are some limitations to this approach. Specifically, to perform massively parallel sorting, the MemSort accelerator must have sufficient capacity to store all numbers within the rows of its memory arrays. Liu et al. [54] indicate that the architecture can sort up to 2^{30} integers, based on their experimental results. However, they do not specify the exact number of arrays, banks, or chips in the system, nor do they discuss what occurs when the dataset exceeds this capacity. If such a limit is reached, the sorting must be performed in multiple passes.

2.2.2. Pattern matching

Apart from VMM, the second most common operation accelerated in CIM is matching, also known as look-up operations. Look-ups in CIM are typically implemented using the CIM-CAM primitive and have a wide range of application domains. In essence, any application that can be reduced to a table look-up can be accelerated with CIM. According to Graves et al. [57], such applications include finite state machines, such as regular expression matching for detecting malicious patterns in network security; tree-based models, such as random forests for classification and regression; and reconfigurable computing for the implementation of arbitrary logic functions.

An abstract example of a pattern-matching memory array is shown in Figure 2.7a. In this design, the memory cells are ternary, storing one of three values: 1, 0, or X (don't care). However, memory cells can be binary or use a higher radix. Once the application to be offloaded is reduced to a lookup table, it can be mapped onto this accelerator by storing a single table entry in each wordline. Data can then be streamed into the input buffer, which performs a parallel match against all entries in the lookup table. A match is detected when current flows from one end of a wordline to the other. Finally, a multiple-match resolver is used in post-processing to return the location of the matching entry.



Match Type	Horizontal only	Vertical only	Horizontal & Vertical
Exact	AND	Gather	AND and Gather
Best	Voting	Comparator	N/A
Threshold	N/A	Gather	N/A

Figure 2.7: (a) Abstract pattern-matching memory array encoding a lookup table in memory cells and performing parallel matching with the input buffer [57, Fig. 1] (b) CAMASim's hierarchical four-layer structure, bank-mat-array-subarray [55, Fig. 2], (c) Horizontal and Vertical Partitioning illustrated [55, Fig. 3a], (d) Different types of matching and their horizontal and vertical merging schemes [55, Fig. 3b].

We now examine an architecture devised by Li et al. [55] as part of their CAMASim simulation framework. Although CAMASim is primarily a design exploration tool for CAM-based accelerators, it also defines a general hierarchical structure shown in Figure 2.7b. This hierarchy supports parallel matching and merging across multiple memory arrays, enabling the manipulation of datasets too large for a single array. Inter-array matching relies on two partitioning schemes—horizontal and vertical—both illustrated in Figure 2.7c. The choice of scheme depends on the application, which may require one of three match types, shown in Figure 2.7d: exact match, where every cell in a row matches the input buffer; best match, where the best approximate matching row to the input buffer is selected; and threshold match, where rows are compared against a specified threshold. Each match type requires different merging operations depending on the partitioning scheme, which in turn may necessitate distinct post-processing circuitry at each hierarchical level.

Overall, search operations are highly suitable for CIM acceleration, as all rows across the accelerator hierarchy can be searched in constant time while minimizing energy usage and cell reprogramming. However, current implementations still face significant challenges. According to Li et al. [55], the motivation behind CAMASim is the lack of an optimal general architecture for pattern matching, which necessitated a design exploration tool. Moreover, new merge schemes are still being defined, and no existing accelerator design can fully address both horizontal and vertical merging for best-match and threshold-match operations.

2.2.3. Convolutional Neural Network inference

Machine learning is the most common domain where CIM is applied, owing to its ability to perform VMMs in constant time using the CIM-Xbar CIM primitive. A prominent CIM application is the inference phase of convolutional neural networks (CNNs), exemplified by accelerators such as ISAAC [52]. We present CNN inference so late because, despite its popularity, it is a substantially more complex application comprising several distinct operations.

Figure 2.8 shows an abstract network architecture diagram for a CNN that performs image classification. CNNs primarily consist of multiple convolutional layers that perform convolution and pooling for feature extraction, followed by fully connected layers that perform linear operations for classification. Figure 2.9 shows that to optimize the CNN, additional operations can be added after the primary layer operations, such as batch normalization to improve training speed and stability, and an activation function to introduce non-linearity and enable the network to learn more intricate patterns.

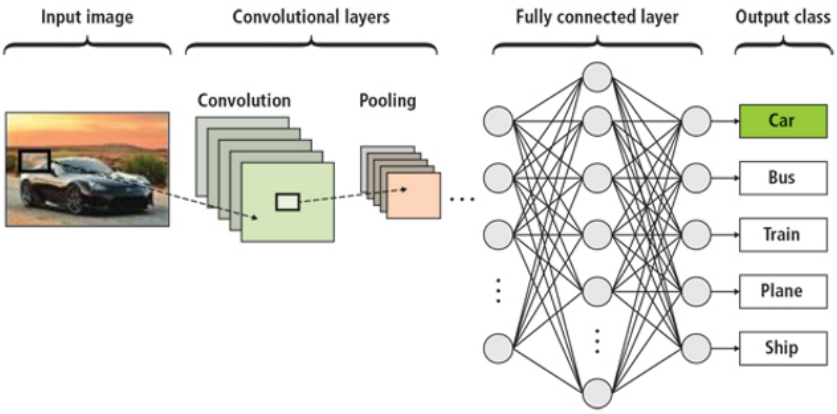


Figure 2.8: Abstract network architecture diagram for a CNN that performs image classification [58] ©NVIDIA.

A particularly suitable subclass of CNNs for CIM is the binarized CNN, or Binary Neural Network (BNN), shown in Figure 2.9. Binarizing a network improves performance and efficiency at the cost of some accuracy by representing both weights and activations in convolutional and linear operations using the `sign` function (Eq (2.1)). These operations therefore reduce to binary-precision VMMs that map naturally onto binary CIM-Xbars, avoiding the complexity of higher-precision memory cells. An example implementation for CIM is CIM-Explorer [56], a toolkit for BNN and Ternary Neural Network (TNN) design-space exploration on CIM-Xbars. The techniques discussed in this section apply to both full-precision CNNs and their binarized variants.

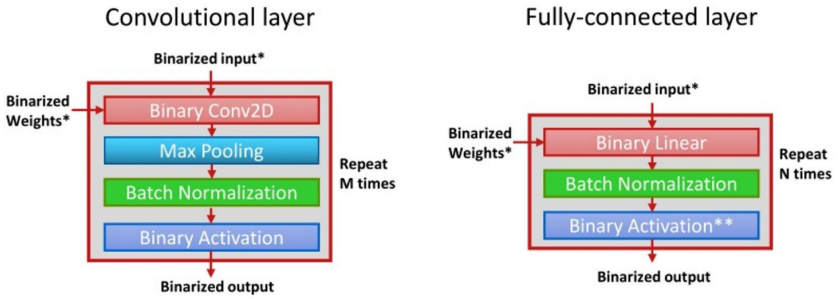


Figure 2.9: Structure of a Binarized CNN as proposed by Courbariaux et al. [59] illustrated by de Bruin [60]. *First and or last layer is not binarized; sometimes the weights are binarized but the input is not.

Mapping CNNs onto CIM hardware is challenging, and even BNNs require careful handling of signed binary values in crossbar cells, as noted by Zaheti [7, p. 44]. A programming model abstracts away such low-level concerns and focuses on expressing the high-level algorithm. Using CIM-Explorer's compilation stack in Figure 2.10 as a reference point, we treat the programming model as a frontend component that reduces applications to matrix–vector multiplication (MVM) operations and exposes compute modes through the functional interface to guide mapping. The mapper then uses these MVM operations and compute modes to reshape the data and map the resulting structures onto the physical crossbars. Thus, because this manuscript focuses on programming models, our examples remain at a high level, and the mapping process lies outside the scope of this work. However, we briefly discuss mapping in Section 5.5.

$$\text{sign}(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases} \tag{2.1}$$

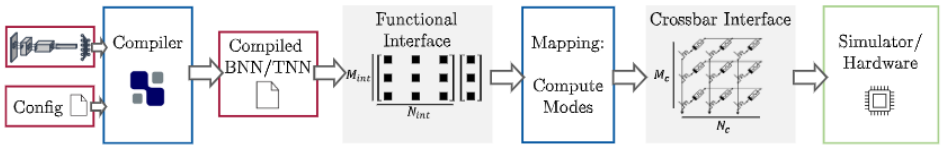


Figure 2.10: The interfaces of the CIM-Explorer toolkit that separates compilation and mapping.[56, Fig. 3]

CNN operations include convolution, linear operations, pooling, batch normalization, and activation. To analyze how these operations are realized in a CIM accelerator, we take a closer look at ISAAC [52]. Following ISAAC's design choices, we present examples that also use max pooling and the sigmoid activation function. In addition, ISAAC omits batch normalization, noting that it is difficult to adapt to crossbars and not strictly necessary. For completeness, we also describe how the binarized variants of these operations differ from their full-precision counterparts.

The most complex operation is convolution as it requires significant rearranging. Pelke et al. [56] show how its variant `conv2d_nhwc` expressed using Eq (2.2) can be decomposed into parallel MVMs on CIM-Xbars using loop transformations. A simplified example is shown in Figure 2.11a. The operation takes an input feature map (IFM), a set of filters, and a stride. The IFM has four dimensions—input width (i_w), input height (i_h), input channels (i_c), and batch size. For simplicity, we assume a batch size and strides (s_h, s_w) of one. The filter or kernel (K) tensor also has four dimensions: kernel width (k_w), kernel height (k_h), kernel channels (k_c, equal to i_c), and output channels (o_c). Assuming a stride of one then the filter slides over the IFM (on the colored squares indicating sliding-window positions) to produce the output feature map (OFM) shown.

$$\text{OFM}(h, w, o_c) = \sum_{i=0}^{k_h-1} \sum_{j=0}^{k_w-1} \sum_{c=0}^{i_c-1} K(i, j, c, o_c)\ \text{IFM}(s_h h + i,\ s_w w + j,\ c) \tag{2.2}$$

To execute this convolution on CIM-Xbars, the IFM and filters are reshaped using the `img2col` technique into the matrix-matrix multiplication (MMM) shown in Figure 2.11b. The filters are converted

into a matrix by merging all dimensions except o_c, while the IFM's sliding windows are flattened into column vectors and merged with the i_c dimension. As shown in Figure 2.11b, the MMM produces the OFM of Figure 2.11a but with flattened output width (o_w) and height (o_h) dimensions.

The MMM shown in Figure 2.11b can be scheduled on the hardware in several different ways, a flexibility that the programming model is intended to hint at. Figure 2.11c illustrates a schedule where the entire filter tensor is encoded on a single CIM-Xbar and each IFM column vector is applied as the input sequentially. However, such a large CIM-Xbar may not exist in practice. In that case, the filter tensor could be distributed across multiple smaller CIM-Xbars, as shown in Figure 2.11d, requiring the accumulation of partial results (purple and pink) to obtain the final OFM elements. Many other schedules are possible, such as exploiting parallelism to execute all IFM column vectors simultaneously.

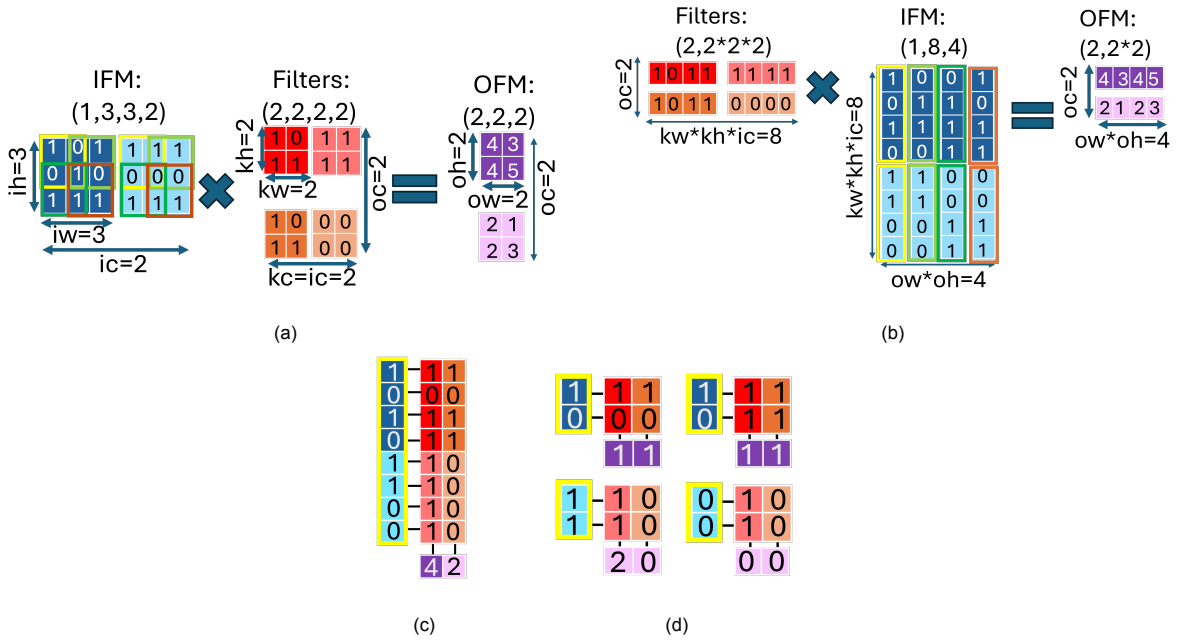


Figure 2.11: (a) Binary `conv2d_nhwc` example with a single-batch IFM and a stride of one. Colored squares on IFM indicate sliding-window positions. (b) `img2col` transformation of Figure 2.11a, resulting in an MMM where filters form the first matrix and the IFM is reshaped into column vectors (c) Illustration of executing the yellow column vector from Figure 2.11b on a single crossbar. (d) Illustration of the VMM from Figure 2.11c distributed across multiple crossbars, requiring accumulation of the purple and pink partial results using addition.

An example fully connected layer is shown in Figure 2.12a. It can be described by Eq (2.3), where x_i are the inputs neurons, W_{ij} are the weights, b_i are the biases and each output y_j is produced by a separate linear operation. This can be then reshaped into the MVM shown in Figure 2.12b and mapped on a large enough CIM-Xbar shown in Figure 2.12c. Of course, if no CIM-Xbar is large enough to hold the entire matrix, the computation can be split across multiple smaller CIM-Xbars, with post-processing used to aggregate the partial results.

$$y_i = \sum_{j=1}^{n} W_{ij} x_j + b_i \tag{2.3}$$

Both convolution and linear operations reduce to MVMs. Binarization just replaces the multiply and accumulate operations done within a CIM-Xbar with XOR and popcount. This leads to a subtle definitional distinction: Zahedi [7] classifies a CIM-Xbar performing XOR instead of MAC as a binarized implementation of a CIM-Xbar, whereas Reis et al. [1] definitions are closer to a CIM-ALU. Since both interpretations are valid, we treat CNN inference as an application that can be executed on either a CIM-Xbar or a CIM-ALU in Table 2.2.

Looking at ISAAC's architecture in Figure 2.13, we observe a hierarchy of CIM-Xbars along with specialized units for max pooling and the sigmoid activation function. Max pooling performs a reduction using the max operation on the outputs of the CIM-Xbars, and the result is then passed to the sigmoid

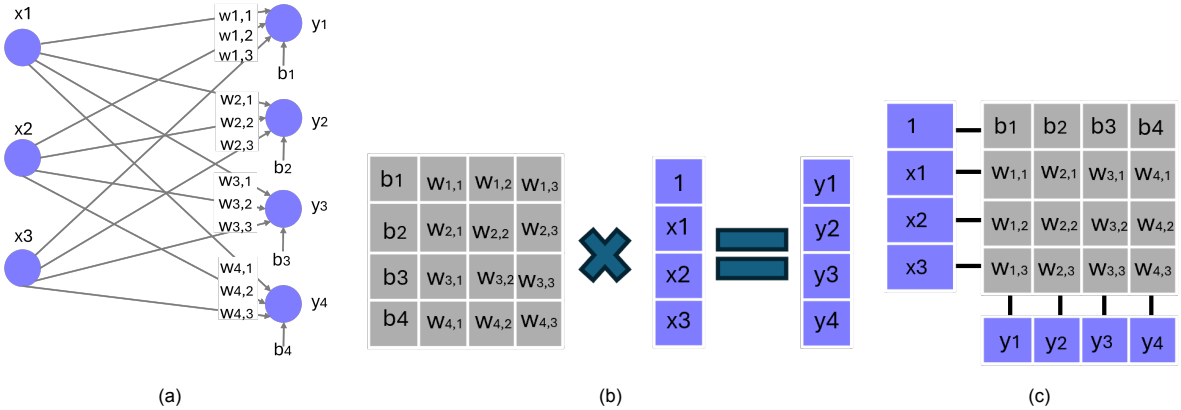


Figure 2.12: (a) Example fully connected layer that performs multiple linear operations in parallel. It consist of three input neurons (`xi`), four output neurons (`yj`) and weights (`wij`). (b) Conversion of Figure 2.12a into a MVM. (c) Conversion of Figure 2.12b into a VMM and placed on a CIM-Xbar with arbitrary precision cells.

unit. In our example recreation of this application for the proposed CIM programming model, we treat both units as black boxes, as their internal workings are irrelevant.

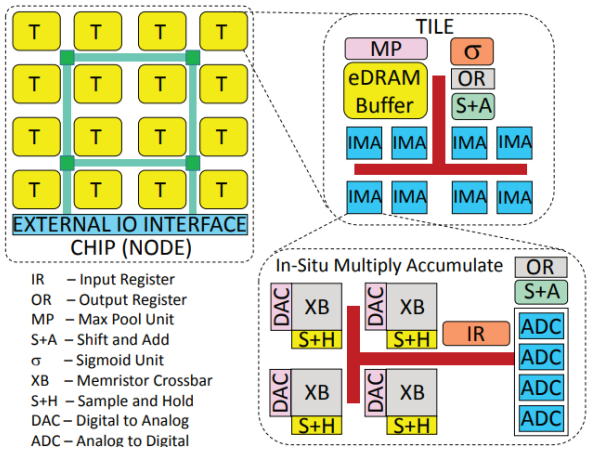


Figure 2.13: ISAAC architecture hierarchy[52, Fig. 2]

2.3. Overview of programming models

Before proposing a new programming model, we first clarify what a programming model is and examine examples from domains more mature than CIM. Hence, Section 2.3.1 defines the fundamental concepts underlying programming models, including their definition, core components, and design trade-offs. Subsequently, Sections 2.3.2 to 2.3.6 survey both programming models and related research, which we later analyze to provide insights relevant to CIM programming models.

2.3.1. Fundamental concepts

In the literature, the term 'programming model' is often used without a precise or universally accepted definition. To remove ambiguity we provide our own set of definitions in Figure 2.14 based on Balaji's [36, p.404] decomposition of OpenCL. Later, we use this definition to construct our own CIM programming model in Chapter 4.

A programming model is essential because it provides a framework for understanding and implementing software on computer hardware. Without such a model, programmers face two alternatives. They can either program a system directly through its instruction set architecture (ISA) or rely on a compiler that abstracts the underlying machine logic entirely through high-level function calls. The first approach demands extensive knowledge of computer architecture and produces large, complex,

Definition: A programming model is a collection of **high-level** abstractions built on top of a programmable computer model, designed to expose specific computational or architectural properties to the programmer.

- **Purpose:** It enables intuitive development of software by summarizing essential functionality to the programmer, without requiring the same depth of knowledge as the hardware designer.
- **Implementation:** A programming model is often realized through the addition of syntax, domain-specific semantics, and programming paradigms that govern the structure of written programs, forming what is commonly referred to as a DSL.

Subdefinition: A computer model is an abstraction over a computing system that defines both its underlying structure and its behavior. It consist of the following interconnected components:

- Platform model: Abstracts away the underlying hardware details and variations across different implementations, presenting a unified and consistent architecture.
- Execution model: Defines how instructions are scheduled and executed on a computer's hardware, enabling programmers to predict a program's behavior by mentally simulating its execution.
- Memory model: Describes the different memory regions within the system and the dataflow of execution such as the inter-thread interaction within memory and sharing of variables.
- Runtime system: Dynamically implements the rules defined by the memory and execution models during program execution.

Figure 2.14: Our programming model definition, adapted from Balaji's [36, p. 404] decomposition of OpenCL.

and hardware-specific programs that, when written correctly, can achieve optimal performance on the target system. The second approach requires no awareness of hardware details and, by leveraging higher levels of abstraction, yields shorter and simpler code that can be more easily ported across different platforms. However, designing such a compiler—particularly one that works across diverse architectures—is highly complex and often comes at the cost of reduced performance. In this sense, both approaches involve trade-offs, while a programming model seeks to strike a balance between them, offering a practical middle ground.

Even so, the design of a programming model inevitably involves a series of choices guided by the goals and priorities of its creator. As Balaji notes [36, Preface], **an ideal programming model embodies four key qualities: (1) productivity, (2) portability, (3) performance, and (4) expressiveness**. In theory, this means that it should be (1) easy to use and allow developers to work efficiently; (2) flexible enough to operate across diverse platforms; (3) capable of delivering performance and resource utilization comparable to hand-optimized code; and (4) expressive enough to represent a wide range of algorithms in a natural way. In reality, however, these qualities often conflict with one another, making the design of a programming model a matter of carefully balancing competing priorities.

For our CIM programming model, certain constraints apply. Specifically, we cannot evaluate performance and portability, as this would require access to multiple programmable accelerators which are currently scarce, whether simulated or physical. Even if that were not the case, time limitations imposed on this work prevented the development of the necessary compilers for each architecture to perform benchmarking, as well as a runtime system to complete the programming model's definition.

2.3.2. Parallel computers

Next, we turn to the study of parallel computers for two main reasons. First, CIM accelerators inherently exploit parallelism to achieve higher performance and can therefore be regarded as parallel computers. Second, the evolution of parallel computers underwent a similarly convoluted period to what CIM accelerators face today, characterized by multiple competing designs and implementations, along with the central question of whether a single computer model could unify them all. By studying parallel computer models, our goal is to draw insights and make informed predictions about the future of CIM computing and programming.

Parallelism has been a longstanding topic of debate in computers because it enables significant performance gains over sequential computers. At the same time, it raised the fundamental question of how to effectively implement parallelism. Multiple efforts were made to classify different parallel

computer implementations in an effort to unify them under a common vocabulary. In our context, a clear classification of parallel computers is useful for determining which programming models align most closely with the forms of parallelism exhibited in CIM architectures.

M.J. Flynn's original taxonomy [61] standardized the classification of computers into four categories based on the number of instruction and data streams shown in Figure 2.15. As a result, most students today are familiar with two of these categories and their abbreviations: Single-Instruction-Multiple-Data (SIMD) and Multiple-Instruction-Multiple-Data (MIMD). However, this classification has been updated multiple times over the years. First, in 1972, M.J Flynn [15] refined the taxonomy further by introducing subcategories within SIMD, based on how multiple processing units (PUs) are utilized. These include the `Array`, `Pipelined`, and `Associative` processors, which are discussed in more detail below. Then in 1988s, E.E. Johnson [62] further categorized MIMD processors into four categories based on their memory structure and their communication mechanism also shown in Figure 2.15. Later in the 1990s, R. Duncan [63] offered modifications to Flynn's taxonomy to include more complex cases shown in Figure 2.16.

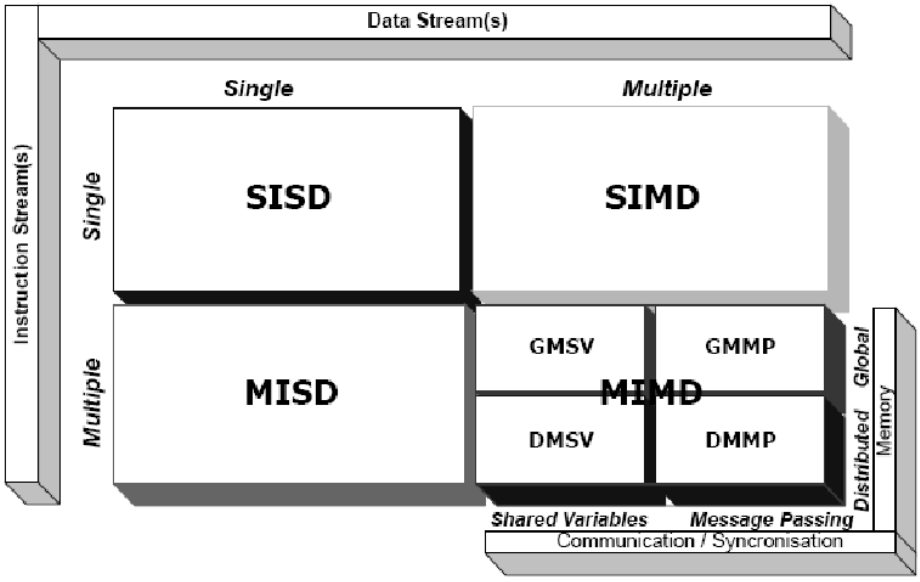


Figure 2.15: Illustration of M.J. Flynn's taxonomy [61]. The MIMD subcategories were added later by E.E. Johnson [62].

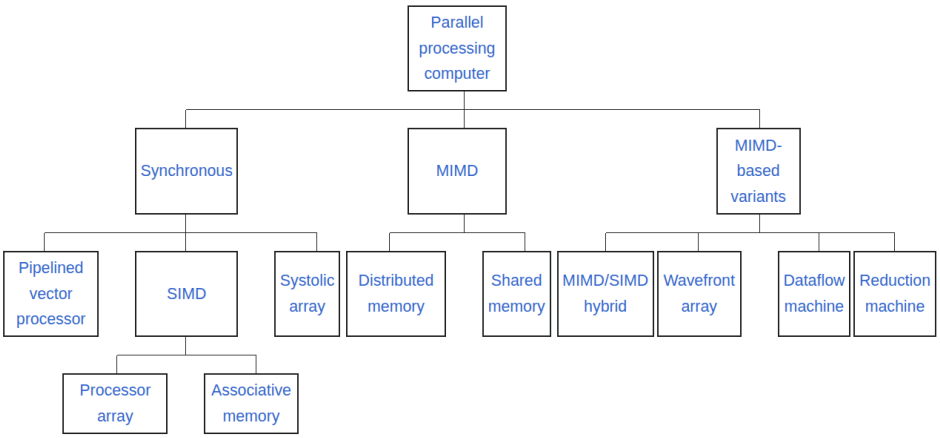


Figure 2.16: Wikipedia illustration of R. Duncan's modifications to M.J. Flynn's taxonomy [64].

Below we provide a short description of each kind of processor subcategory:

M.J. Flynn's SIMD proposal 1972 [15]:

- `Array processor`: Each independent PU has its own local memory, but all are controlled by a single central control unit;
- `Pipelined processor`: All PUs share access to a central memory, but are also specialized for a particular function so they receive a fragment of the incoming data. Data is streamed at a constant rate into each PU that processes them into pipeline stages and write results back to the memory;
- `Associative processor`: Each PU independently decides, based on its local data, whether to execute or skip an operation.

E.E. Johnson's MIMD additions 1988 [62]

- `Shared Memory (GMSV)`: PUs have access to a global memory and communication is achieved using the shared variable memory reference mechanism. Having shared variables is advantageous for expressing parallelism but disadvantages include requiring cache coherency protocols and memory contention as all PUs share the same interconnection to the global memory;
- `Message Passing (DMMP)`: The system's memory is distributed across processors and communication is achieved by passing messages between them. A disadvantage is that programmers must deal with data distribution;
- `Hybrid (DMSV)`: Distributed memory architecture that uses the shared variable programming model. Requires synchronization mechanisms similar to GMSV but avoids the memory contention bottleneck;
- `GMMP`: Global memory architecture that uses the message passing memory reference mechanism. Employ isolated virtual address spaces.

R. Duncan's suggested modifications 1990 [63]

- `Pipelined vector processor`: Like a pipelined processor but using vector units, applying one operation to many data points per stage;
- `MIMD/SIMD hybrids`: MIMD machines that can, to some extent, be programmed in the same manner as SIMD machines;
- `Systolic arrays`: Pipelined multiprocessor variant where data flows synchronously from memory to a regularly shaped processor network. Processors communicate by sending partial results to each other. Requires a global clock and explicit delays between processors;
- `Wavefront array`: Systolic arrays combined with asynchronous dataflow;
- `Dataflow machine`: Executes Data-Flow Graphs (DFGs) to exploit task-level parallelism. Each node fires when all inputs are available, consumes them, and produces outputs for subsequent nodes [20, p. 6];
- `Reduction machine`: Demand-driven dataflow machines. A node requests data from its environemnt. Once the environment responds with data it is then processed by the node and output values are produced [20, p. 6].

Looking at Figures 2.15 and 2.16, our initial goal was to categorize CIM devices within a single parallel-computer class and, from there, identify the most appropriate programming model for that class. However, upon closer examination, it became evident that CIM computers cannot be confined to a single category. Different implementations exhibit characteristics that overlap with multiple paradigms, including pipelined vector processors, systolic arrays, and dataflow machines. Even GPUs cannot be precisely classified within this taxonomy: while warps operate as SIMD units executing the same instruction, they also share associative processor–like behavior, as their computations can vary slightly based on local data. Moreover, modern GPUs increasingly incorporate MIMD capabilities [65].

Although CIM cannot be placed neatly within a single parallel-computer category, this observation directed our attention toward a more general computational model. As noted in Section 1.1.2, the parallel RAM (PRAM) model is unsuitable due to contention and unrealistic unit-time memory assumptions. Instead, we adopt the Candidate Type Architecture (CTA) model, which comprises multiple sequential processors with fast local memory and slower remote-memory access, without imposing constraints on the interconnection topology.

As discussed in Section 1.1.3, the CTA model aligns well with CIM architectures: each CTA processing element corresponds naturally to a CIM macro performing computation within its own local memory, while communication between macros mirrors the CTA interconnection network. The CTA's clear distinction between local and remote access, together with its flexibility regarding network organization, makes it an appropriate abstraction for a wide range of CIM accelerator designs. Thus, using the CTA model as a blueprint, we developed the platform model for our programming model in Section 4.1.2.

2.3.3. OpenCL

OpenCL is an industry standard framework for programming heterogeneous systems [36]. It is considered in this work because a CIM accelerator can function as a coprocessor connected to a general-purpose host, thereby forming a heterogeneous system. This section summarizes the platform, memory, execution, and programming models defined by OpenCL.

The platform and memory models of OpenCL are illustrated in Figure 2.17a and Figure 2.17b respectively. A host system is connected to one or more devices, each composed of a two-tier hierarchy of compute units that contain multiple processing elements. In addition, OpenCL defines several distinct memory regions: host memory, device memory, local memory, and private memory. Device memory is further divided into global and constant memory. The global memory is accessible for both read and write operations by the host and all processing elements, while the constant memory is allocated by the host but is read-only and accessible only to the device. Finally, local memory is shared among processing elements within the same compute unit, and private memory is exclusive to each individual processing element.

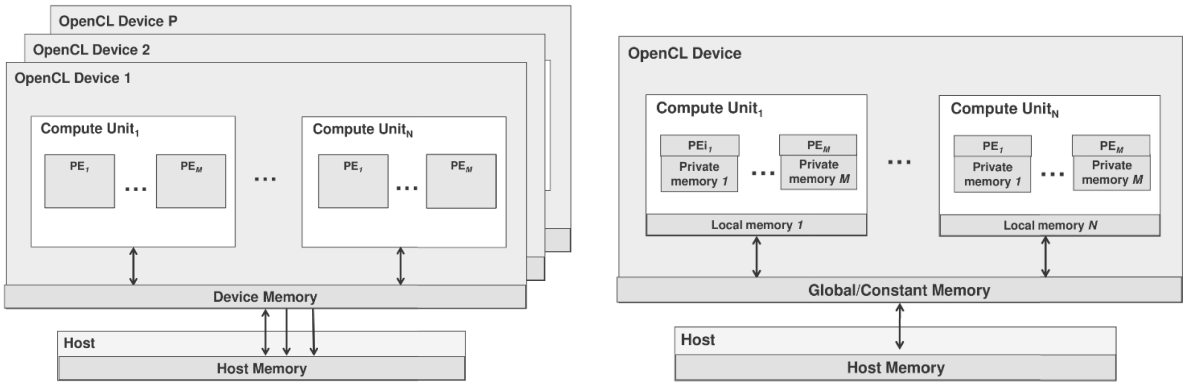


Figure 2.17: (a) Illustrations of OpenCL's platform model [36, Figure 16.1] (b) Illustration of OpenCL's memory model [36, Figure 16.3]

.

OpenCL's execution model enables the exploitation of both data-level and task-level parallelism. Programs define functions, called kernels, which are written by the programmer and offloaded from the host to an OpenCL device for execution. Kernels exploit data-level parallelism through the N-dimensional (ND) range abstraction. For each kernel launch, an ND index space of fixed dimensions is defined, and each processing element executes the kernel on a distinct spatial coordinate within this space—referred to as a work-item. Work-items are grouped into work-groups, which logically execute concurrently. To optimize performance, programmers must ensure that work-items are evenly distributed across processing elements and compute units. Otherwise, the runtime scheduler may fail to fully exploit the device's available parallelism, leaving some processing elements idle and reducing overall throughput.

Task-level parallelism in OpenCL is achieved through the `command queue` abstraction that represents a schedule. Before execution, kernels are enqueued into a command queue associated with a specific device. Programmers can either use `event` objects to form dependencies between kernels in the same command queue, enabling pipelined execution, or create multiple command queues that execute concurrently, allowing overlap of computation and communication to further improve performance. ND ranges and command queues make up OpenCL's programming model.

2.3.4. Halide

Halide [48] is an embedded domain-specific language (DSL) in C++, intended for image and tensor processing. We chose to investigate Halide as the CIM-Explorer [56] toolkit analyzed in Section 3.1 extends the Tensor Virtual Machine (TVM) compiler [66] that adopts ideas from Halide. While CIM-Explorer is specifically designed for Neural Networks we aimed to evaluate if Halide's ideas can be adopted for other CIM applications.

Halide is fundamentally a framework for defining high-performance image processing pipelines on multicore processors. It achieves this by cleanly separating the algorithm specification of a program from its scheduling decisions. This separation allows programmers to express what computation should be performed independently of how it is executed. Scheduling functions can then be applied to tailor execution to a specific device, optimizing for a particular type of parallelism, improving temporal and spatial locality through cache-aware scheduling, and managing redundancy where recomputation is more efficient than memory access.

We illustrate Halide's notation using the example shown in Listing 2.1. Halide primarily relies on two key abstractions: Funcs and Vars. A `Func` is a functor that represents a function defined over an image or multidimensional data domain. Multiple Funcs can be chained together to form image-processing pipelines, where each `Func` represents an intermediate stage in the computation. The pipeline is constructed by defining one `Func` (in this case, bh) in terms of another (bv), forming an acyclic dataflow graph internally. A `Var` is a symbolic variable that, by itself, has no inherent meaning but is used in conjunction with Funcs to define indexing or iteration dimensions. In the example, both `x` and `y` are used to specify that the blurring operation should be applied horizontally (left and right) and vertically (top and bottom), respectively.

As mentioned, a program in Halide conceptually consists of two parts: the algorithm and the schedule. While it is common in introductory examples to specify the entire algorithm first and then apply scheduling, this is only a convention used to illustrate the separation. Halide does not require the algorithm to be fully defined before scheduling begins, since scheduling directives can be applied at any point after a Func is defined. However, in our motivational examples in Chapter 4 we keep this convention in order to clearly illustrate the separation between algorithm and schedule.

When the algorithm part finishes defining all Funcs, no computation is executed yet, because Halide uses lazy evaluation to construct the computation graph. Actual execution begins only when the `realize` function is called, at which point Halide generates the corresponding nested C++ loops that implement the described algorithm, which is a form of meta-programming. The schedule allows the programmer to influence these generated implicit loops through declarative loop transformation directives applied to Vars, often parameterized by integers.

Common directives include `tile`, `vectorize`, `parallel`, and `compute_at`, as illustrated in Figure 2.18. Specifically, `vectorize` executes iterations of a given variable using SIMD instructions of a specified width; `parallel` marks a loop for multithreading; `compute_at` specifies the loop level at which a function is recomputed, controlling locality and the trade-off between recomputation and storage; and `tile` restructures nested loops into smaller blocks to improve data locality.

2.3.5. Legion

Legion [38] is a data-centric parallel programming system designed for high-performance execution on heterogeneous and distributed architectures, which aligns well with the target platform of CIM accelerators. Legion is particularly interesting because it allows programmers to explicitly define applications based on program data properties, such as dependencies and locality, enabling the automatic extraction of both task-level and data-level parallelism.

Legion is built on three core principles that define how programs should be structured to achieve optimal performance on its targeted systems. Each principle is embodied by a corresponding abstraction. First, logical regions allow program data properties to be explicitly declared through high-level abstractions. Second, a hierarchical task tree operates on these logical regions implicitly expressing parallelism and data movement. Third, the mapping interface ensures that a program remains independent of the specific details of how it is mapped onto the underlying hardware. The following paragraphs describe each abstraction in detail.

Logical regions can be viewed as an extension of the ND-range abstraction introduced in Section 2.3.3. Specifically, they represent N-dimensional data structures that can be shared among different processes, allowing programmers to explicitly characterize an application's data. These char-

```
1  Func in = ...
2  Func bh, bv;
3  Var x, y, xi, yi;
4  // The algorithm
5  bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))
       /3;
6  bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))
       /3;
7  // The schedule
8  bv.tile(x, y, xi, yi, 256, 32)
9   .vectorize(xi, 8).parallel(y);
10 bh.compute_at(bv, x)
11  .vectorize(x, 8);
12 bv.realize({imgDimX, imgDimY});
```

Listing 2.1: Halide code for a blur application with two pipeline stages, one blurring horizontally and one vertically

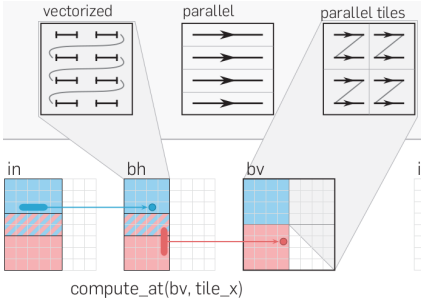


Figure 2.18: Corresponding scheduling directives visualized.

Figure 2.19: Halide blur application and its schedule[48, Fig. 1c].

acterizations include specifying dependencies between variables, assigning read-only or write-only privileges to a processing accessing a variable, and ensuring coherence among concurrent accesses. As an example, consider the DAXPY operation shown in Eq (2.4), where x, y, and z are vectors and a is a scalar. This operation can be represented using the logical regions illustrated in Figure 2.20a. For each variable in the equation, a field space is created, and for each data point, an index is generated to form an index space. In this case, dependencies exist only among entries within the same row, whereas the process computing z requires only read privileges for x and y. By leveraging this information, Legion extracts optimal parallelism while preserving the algorithm's correctness.

The task tree abstraction is a dataflow graph that captures the dependencies between logical regions forming independent tasks. Legion uses a deferred execution model to asynchronously analyze, optimize and execute tasks with all the available parallelism and dataflow coordination implicitly extracted. For the DAXPY example shown in Figure 2.20a, the resulting task tree is shown Figure 2.20b. It shows four groups of tasks: one that initializes the x and y field spaces, one that performs the computation, one that validates the result, and one that deallocates the memory used for the field spaces.

The idea behind the mapping interface is to separate the algorithm from its scheduling on the machine, as the mapping should be independent of the program's correctness. Hence, programs are expressed in terms of logical regions, which are later mapped to physical regions. The mapping interface is an abstraction that programmers must implement for their application on a particular architecture. This separation facilitates easier porting and performance tuning. For example, the mapping can specify where each vector in Eq (2.4) is stored or how the tasks, shown in Figure 2.20b, are distributed across the available processors.

$$\mathbf{z} = a\mathbf{x} + \mathbf{y} \tag{2.4}$$

2.3.6. Apache Spark

We now turn to distributed computing systems, particularly cluster computing, because of their structural similarities to CIM architectures, including multiple interconnected processing and memory units as well as challenges in concurrency, data distribution, and inter-element communication. Our aim is to examine the programming model of Apache Spark [68], a widely used distributed computing framework, and to evaluate its potential applicability to CIM programming.

We begin by first defining a cluster's platform model as explained by Lin et al [3, p.39]. A cluster is a specific type of distributed computing system composed of homogeneous nodes, each consisting of a small number of processors, interconnected through a high-speed local network. Nodes have access to their local memory and inter-node communication is carried out via message passing. In addition, a single node is referred to as the driver as it assumes the responsibility of interfacing with the user and initiating program execution.

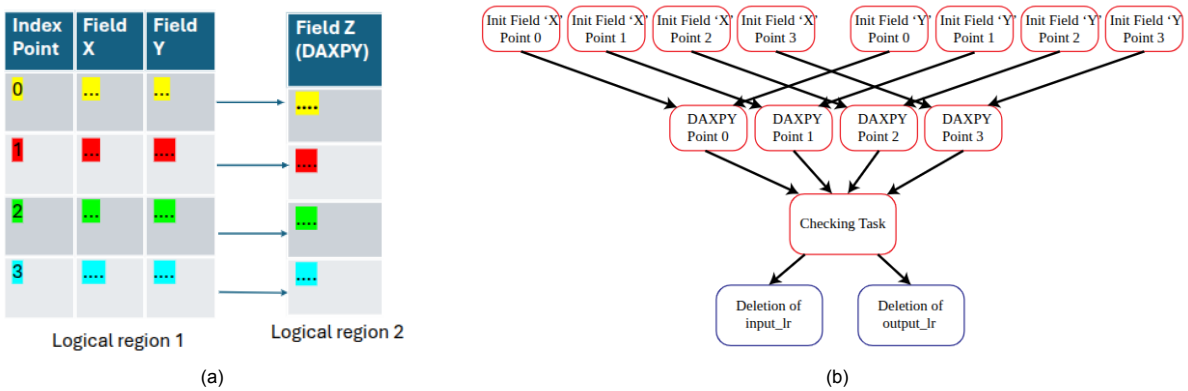


Figure 2.20: (a) Illustration of logical regions for DAXPY example. (b) The task tree for the DAXPY example [67].

Apache Spark is a multi-language analytics engine designed for large-scale data processing on multi-node systems. It provides a functional programming interface that promotes fault tolerance, scalability, and implicit parallelism. This is achieved through Spark's programming model, whose core abstraction is the Resilient Distributed Dataset (RDD). An RDD is a read-only collection of objects that is partitioned and distributed across multiple nodes, enabling parallel computation. Specifically, the driver program sends RDD partitions, along with the transformations to be applied, to worker nodes. Each node then applies these transformations to its partition and returns the results to the driver.

In the event of a fault that prevents a result from reaching the driver, Spark relies on two key properties of RDDs to ensure fault recovery. First, RDDs are immutable, meaning that every transformation creates a new RDD. This allows intermediate datasets to be reused to resume computation from the last successful stage. Second, each RDD maintains a lineage, a record of the transformations applied to the original dataset. Using this lineage information, Spark can reconstruct lost partitions by replaying the transformations on the source data. Additionally, users can modify scheduling using function calls such as the preferred location for each partition within the memory hierarchy of the destination nodes, improving data locality and performance.

In addition, Spark provides two types of shared variables. First, broadcast variables allow the user to define large read-only data objects that are sent to each worker only once, thereby reducing communication overhead. Second, accumulators support associative operations that can be updated by the workers but read only by the driver, making them useful for tasks such as implementing counters.

Apache Spark focuses heavily on resilience, a concern that does not arise at the programming level of CIM and therefore makes Spark largely incompatible with CIM. However, the idea of introducing immutability in our proposed programming model not for resilience but for energy efficiency, as discussed in Section 2.1.1, was inspired by Spark. In addition, the idea of using shared variable types with different communication patterns inspired the memory objects discussed in Section 4.1.3.

2.3.7. Dataflow programming languages

We examine dataflow computers and languages because several CIM compilers and frameworks discussed in Chapter 3 adopt a dataflow execution model. Moreover, Milojicic et al. [42] note that many applications that benefit from CIM naturally align with dataflow programming principles. Understanding dataflow computing helps determine whether new CIM programming models should adopt a dataflow execution model or incorporate other abstractions from the dataflow domain.

The core idea behind dataflow computers is that programs are represented as directed graphs where execution is driven by data availability rather than a sequential instruction order. These Dataflow Graphs (DFGs) explicitly capture how data is produced, consumed, and transformed through dependencies between operations. Execution occurs by mapping operations (nodes) onto hardware resources, where each node consumes input data and produces outputs that trigger subsequent computations. Each value flowing between nodes is represented as a token, while the communication channels between nodes are referred to as arcs. Figure 2.21 illustrates a simple dataflow graph for computing `Z`. In this graph, circles represent operators, `X` and `Y` denote input sources, and rectangles correspond to constants. The execution of a DFG depends on the underlying execution model.

There are two commonly discussed execution models for dataflow systems. In the data-driven execution model, used in multiscale dataflow computing [29], a node fires (performs its operation) only when an instance of each required input has arrived at its input arcs. Once triggered, the node consumes the input tokens, processes them, and produces output tokens that may enable downstream nodes. Alternatively, languages such as Lucid [69] employ a demand-driven execution model, where computation is initiated by the need for a specific result rather than by operand availability. In this model, execution conceptually flows backward: it begins when a node requests data from its environment through its input arcs, prompting upstream nodes to request and compute the data they require. When the requested data is produced, it then flows to downstream nodes, which process it and generate their outputs. Applying either execution model to Figure 2.21b would yield the same result. According to Wadge [70], the tradeoff is that demand driven dataflow is more general in terms of expressiveness but it has a higher overhead in keeping track of requests.

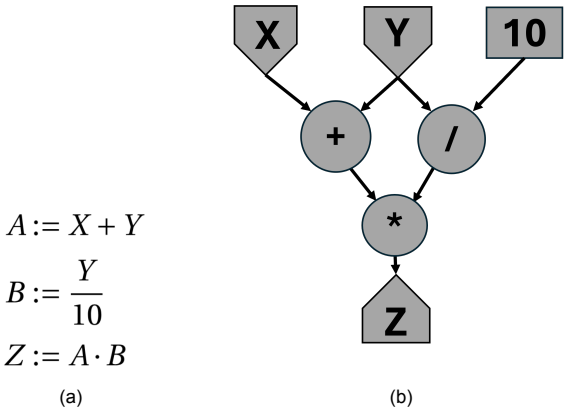


Figure 2.21: (a) Simple program. (b) Equivalent dataflow graph, where circles represent operators (nodes), `X`, `Y`, and `Z` denote inputs and outputs, and rectangles represent constants.

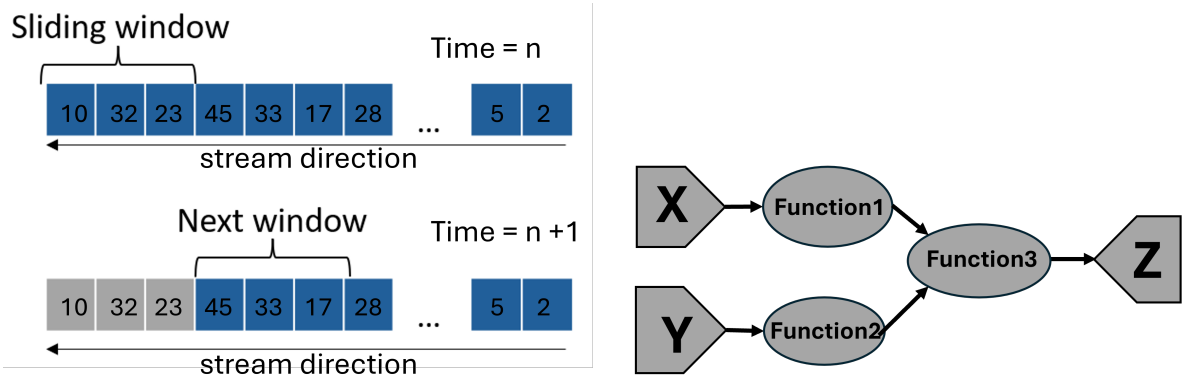
Dataflow computing can offer significant advantages over conventional control-flow computing. Traditional machines impose a strict ordering of program instructions, making parallelism implicit and often difficult to extract. As a result, compilers must expend considerable effort to uncover parallelism and schedule it while preserving program correctness. In contrast, dataflow computing enforces ordering constraints only through true data dependencies between values, which are explicitly represented in a DFG. This makes parallelism explicit and enables fine-grained exploitation of concurrency.

According to Ragan et al. [20, p. 10], dataflow languages are not exclusive to dataflow computers and can be regarded as a subclass of functional programming languages. They share several core characteristics with functional languages: they avoid side effects, ensuring that operations do not alter external state and depend solely on their inputs; they exhibit locality of effect, meaning that each operation influences only the data it directly processes; they enforce single assignment of variables, maintaining immutability and simplifying dependency tracking; and they lack history sensitivity in procedures, such that function invocations are stateless and do not retain information across calls. These properties also necessitate non-traditional approaches to iteration, often relying on recursion or specialized iteration constructs rather than mutable loop variables. The key distinction from conventional functional programming is that scheduling in dataflow languages is determined by data dependencies in the generated DFG.

Dataflow programming can provide CIM programming models with valuable abstractions that extend beyond dataflow graphs and execution models. One of the most prominent is the stream abstraction, which represents a (possibly infinite) sequence of data tokens of the same type. Streams are immutable and align well with the principle of avoiding global state. In Figure 2.21b, `X`, `Y`, and `Z` are all examples of data streams. As illustrated in Figure 2.22a, a stream evolves over time as new tokens arrive. Streams can also be processed in windows, allowing multiple values to be consumed at once, and may use a stride to determine how computations progress across iterations. Delays can be introduced to enable computations between values from different points in a stream. Additionally, streams may contain absent ($\perp$) values, representing the lack of data output from a node at a particular time step.

An additional abstraction is the separation of computation and coordination. DFGs are highly flex-

ible, as they can express varying levels of granularity. Nodes may represent fine-grained operations, such as the arithmetic in Figure 2.21b, or coarse-grained tasks, such as entire functions in Figure 2.22b. Consequently, many existing frameworks and programming models—such as Legion [38], TVM [66], and MaxCompiler [29]—employ dataflow semantics as a coordination language between tasks. The benefit of the separation is that tasks can be defined and subsequently scheduled independently. At the same time as tasks are purely declarative, different hardware dependent optimizations can be easily applied



(a) (b)

Figure 2.22: (a) A stream of integers starting from the left and ending to the right over two consecutive time instances. The sliding window and the stride are both of size three. (b) Dataflow graph, that shows the coordination between three functions.

3

Analysis of existing programming models

This chapter presents an analysis of existing abstractions, both conventional and CIM-specific, with the goal of identifying essential meta-abstractions that can inform the development of CIM programming models. By examining established work—such as programming models, compilers, and frameworks—and analyzing how they use different abstractions to balance productivity, portability, performance, and expressiveness, we derive a set of desirable meta-abstractions to guide the design of our proposed model. We divide this work into three parts. First, in Section 3.1, we analyze CIM-specific compilers and frameworks. Second, in Section 3.2, we examine relevant work from domains beyond CIM. Finally, Section 3.3 concludes with a list of meta-abstractions and possible corresponding abstractions.

3.1. CIM compilers and frameworks

It is important to learn from prior work on the programming of CIM accelerators to identify which existing ideas can be adopted or adapted in our own contributions. To this end, we have analyzed several compilers and frameworks as summarized in Table 3.1. The table includes four fields: (1) the application domain, indicating whether the tool is restricted to a specific class of applications or supports a specific type of CIM primitive, (2) whether the tool adopts a programming model or is purely API-based, (3) the programming frontend used, and (4) whether the programmer can influence CIM-specific aspects such as resource management. Table 3.1's first row, presents the ideal case for CIM compilers and frameworks. They should be:

- **Domain-agnostic**, enabling the expression of a wide range of applications without being restricted to a specific set of operations;
- **Based on a programming model rather than a fully declarative (API-based) approach**, allowing both computational and scheduling decisions to be influenced by the programmer;
- **Enable programmer control over CIM-specific aspects** such as resource management.

We summarize our observations from Table 3.1. First, almost all of the relevant works target machine learning (ML) or deep neural networks (DNN), as matrix-vector multiplication (MVM) is the dominant operation in these domains and is well suited for CIM acceleration. Efforts that aim to be more general restrict themselves to two CIM primitives—the CIM-XBar and the CIM-ALU—which share a similar structure. Of the surveyed tools, only C4CAM is truly domain-agnostic, as it can compile to all CIM primitive types. Second, the API-driven nature of ML and the complexity of CIM accelerators (see Section 2.1) mean that most tools rely on existing frontends and compilers, avoiding the need for a dedicated programming model altogether. Third, there is no stable programming frontend used across tools with some as low level as C, high level as Pytorch or even higher in the form of a graph using the Open Neural Network Exchange (ONNX) [75] format. Finally, almost none of the tools expose CIM-specific features to the programmer as they treat mapping and scheduling decisions as too complex to be managed manually. We now go into greater detail on each tool in the following paragraphs.

Name	(1) Application domain	(2) PM or API	(3) Frontend	(4) CIM-specific features
Ideal case	domain-agnostic	PM	–	yes
OCC[71]	MVM	API (CIM dialect)	Teckyl	no
CINM[44]	limited CAM support	API (CIM dialect)	Pytorch/Linalg/Tosa	no
C4CAM[72]	domain-agnostic	API (CIM dialect)	TorchScript (Pytorch IR)	no
Polyhedral[73]	DNN (MVM)	API	C	compile arguments
TC-CIM[47]	DNN (MVM)	API	Tensor Comprehensions (Pytorch)	no
TDO-CIM[74]	MVM	API	C++	no
CIM-MLC[45]	DNN (MVM)	API	ONNX	architecture parameters
Co-design[46]	ML	API	ONNX	no
CIM-Explorer[56]	BNN/TNN	PM (Halide based)	Larq	strategies per operation
IMDP[51]	no CAM support	PM	TensorFlow	no
PUMA[50]	ML	PM & API	C++/ONNX	no

Table 3.1: Comparison of CIM compilers and frameworks. The attributes compared are: (1) Any restrictions on application domain, (2) if a programming model (PM) is used or if it is purely API-based, (3) the frontend the user interfaces with (4) whether the user has any control over CIM-specific aspects such as resource management.

OCC [71], **CINM** [44], and **C4CAM** [72] form a series of end-to-end compilers for CIM that leverage multilayer abstractions through MLIR [76]. OCC was originally developed for MVMs, later extended in CINM to support Computing-Near-Memory (CNM) and the CIM-ALU primitive, and finally enhanced in C4CAM to include the CIM-CAM primitive. All of them make use of the CIM dialect, whose operations—shown in Figure 3.1—include buffer manipulation, matrix multiplication, and synchronization. All three compilers call the CIM dialect a programming model, though this is inaccurate by our definition in Section 2.3.1. The CIM dialect is employed by compilers to interface with accelerators and perform hardware-specific optimizations. Since the dialect is never directly exposed to the programmer, we do not consider it to constitute a programming model. In addition, C4CAM introduces a hierarchical hardware abstraction that generalizes CIM-CAM accelerators as shown in Figure 3.2.

Operation	Datatype	Target	Description
cim.write(`%id`, `%matB`)	integer, memref	CIM	Transfer and write the 2D buffer `%matB` to the crossbar of the CIM tile `%id`, synchronously (or blocking).
cim.matmul(`%id`, `%matA`, `%matC`)	integer, memref, memref	CIM	Transfer the 2D buffer `%matA` to the CIM tile `%id`, perform GEMM and store the results in the 2D buffer `%matC`, asynchronously (or non-blocking).
cim.barrier(`%id`)	integer	HOST	Wait for work completion on the CIM tile `%id`.
`%tile` = cim.copyTile(`%mat`, `%row`, `%col`)	memref, memref, index, index	HOST	Allocate a contiguous 2D buffer `%tile` and copy a tile at the position (`%row`, `%col`) from the 2D buffer `%mat`.
cim.storeTile(`%tile`, `%mat`, `%row`, `%col`)	memref, memref, index, index	HOST	Copy the tile `%tile` to the 2D buffer `%mat` at the position (`%row`, `%col`).
cim.accumulate(`%lhs`, `%rhs`)	memref, memref	HOST	Add corresponding elements of the `%rhs` to the `%lhs`.
`%output` = cim.allocDuplicate(`%input`)	memref, memref	HOST	Allocate an empty buffer `%output` which has the same dimensions as the input buffer `%input`.

Figure 3.1: The CIM dialect operations illustrated by OCC[71, Table 1]

Polyhedral [73] is a source-to-source compilation framework for neural networks on CIM systems that is capable of auto-detecting supported operations in programs written in C to be offloaded for acceleration. It accepts compilation arguments that allow the user to specify hardware constraints. These include the number of processing elements and the dimensions of the supported MVM operands.

Both **TC-CIM** [47] and **TDO-CIM** [74] are end-to-end compilation flows based on LoopTactics [77] that automatically identify operations in sequential code suitable for acceleration. These operations include MVM and other variants. Both frameworks assume an accelerator consisting of a single CIM-Xbar. As no programming model is used in any of these works they are of little use to us.

CIM-MLC [45] is a multi-level Neural Network compilation framework designed to be universal across CIM architectures. It accepts neural networks in the ONNX format, applies optimizations, and generates an instruction set architecture (ISA) targeting one of three hardware abstraction layers, depending on the granularity supported by the underlying device. This hierarchy is similar to the one shown in Figure 3.2. However, it is used to abstract CIM accelerators that employ the CIM-Xbar primitive. It consists of three tiers, named from top to bottom: chip–core–crossbar.

Co-design [46] is a software stack for CIM-Xbar accelerators. It is API based as different Neural Network formats can be reduced to the ISA devised in the paper. All of these formats are high level and abstract away the entirety of the hardware. It also employs a hierarchical hardware abstraction, similar to CIM-MLC, but with the hierarchy renamed to tile–core–crossbar.

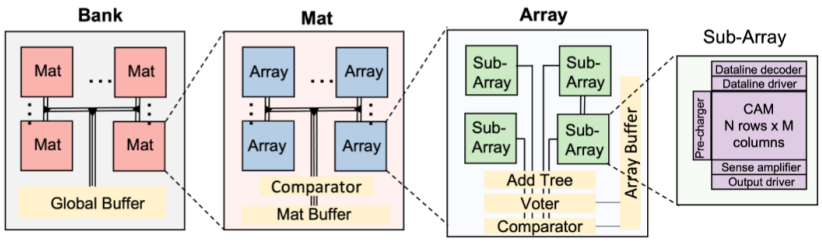


Figure 3.2: Illustration of C4CAM's three-tier hierarchical hardware abstraction [72, Fig. 2], with similar abstractions found in Co-design[46], PUMA[50] and CIM-MLC[45]. The names of each tier might be different across implementations such as Co-design and PUMA that use tile-core-crossbar.

CIM-Explorer [56] is a toolkit for exploring binary and ternary neural networks on CIM-Xbars. It extends the TVM [66] compiler, which was itself inspired by Halide's [48] programming model discussed in Section 2.3.4. Consequently, CIM-Explorer's compiler adopts the principle of deferred execution where instructions are not executed immediately but slowly build an internal representation that is implicitly scheduled. In addition, it separates computation from scheduling, allowing programmers to use scheduling strategies that preserve the logical equivalence of programs while enabling optimizations.

In-Memory Data Parallel Processor (IMDP) [51] proposes a hierarchical architecture for CIM and provides a parallel programming framework that combine concepts from dataflow and vector processors. This is achieved by reusing TensorFlow's programming model, where kernels are used to construct operations that receive and produce multidimensional (ND) data allowing programmers to manage data-level parallelism. Dataflow is then controlled by assembling pipelines of these operations to control task-level parallelism. The resulting Data Flow Graphs (DFGs) are optimized and offloaded for execution. Interestingly, irregular memory structures are restricted by disallowing subscript notation, a choice that simplifies compilation at the cost of expressiveness. However, the framework does not support the CIM-CAM primitive, and its paper provides no references to code examples.

PUMA [50] is a CIM-Xbar architecture with its own compiler that translates high-level code into the PUMA ISA. PUMA is a spatial architecture, similar to dataflow engines, in which each processing unit executes a different set of operations, forming a pipeline. Consequently, a dataflow programming model is adopted, where programmers explicitly construct DFGs using nodes, vector inputs and streams. Alternatively, a neural network in ONNX format can be provided in place of using their programming interface. While conceptually similar to IMDP, PUMA is designed to optimize MVM operations for machine learning, whereas IMDP targets more general-purpose vector operations.

In conclusion, only **CIM-Explorer**, **IMDP**, and **PUMA** implement programming models. Additionally, several frameworks introduce useful hierarchical hardware abstractions. In Table 3.2, we summarize all identified abstractions along with the underlying meta-abstractions they expose to programmers. Broadly, these abstractions fall into one of four conceptual categories: those that explicitly express task or data parallelism, making parallelism easier to detect and exploit; those that enable implicit scheduling by allowing programmers to guide execution without specifying exact ordering; those that support optimization by giving programmers control over how computations are tuned to hardware; and those that promote portability by generalizing across different hardware.

Framework	Abstractions	Underlying meta-abstraction
IMDP[51]	Dataflow execution model Kernels form DFG Kernels can operate on ND data	Implicit scheduling Explicitly express task parallelism Explicitly express data parallelism
PUMA[50]	Dataflow execution model Nodes and streams form DFG Vector inputs to nodes	Implicit scheduling Explicitly express task parallelism Explicitly express data parallelism
CIM-Explorer[56]	Deferred execution model Declarative scheduling directives Separation of algorithm from scheduling	Implicit scheduling Optimization Hardware generalization
CIM-MLC[45], Co-design[46], C4CAM[72]	Hierarchical hardware abstraction	Hardware generalization

Table 3.2: Abstractions from existing CIM frameworks and the underlying meta-abstraction they expose to programmers.

3.2. Conventional frameworks and abstractions

In Section 2.3, several frameworks were presented from domains other than CIM. In this section, we analyze each of them to extract insights relevant to CIM programming models. In particular, we focus on the abstractions these tools employ and the underlying properties they expose to the programmer.

In Table 3.3, we list the abstractions employed by each framework and the corresponding meta-abstraction they expose to the programmer. Similar to Table 3.2 all abstractions fall into one of four categories: expressing data/task parallelism, implicit scheduling, optimization and hardware generalization. For the rest of the section we discuss each framework's abstractions individually considering if each framework could be reused in a CIM context.

Framework	Domain	Abstractions	Underlying meta-abstraction
-	Parallel computing [3]	CTA model	Hardware generalization
OpenCL[36]	Heterogeneous computing	Platform model ND ranges Command queues	Hardware generalization Explicitly express data parallelism Explicitly express task parallelism
Halide[48]	Image processing	Deferred execution model Declarative scheduling directives Separation of algorithm from scheduling	Implicit scheduling Optimization Hardware generalization
Apache Spark[68]	Distributed computing	Deferred execution model Shared variable types	Implicit scheduling Optimization
Legion[38]	High performance computing	Deferred execution model (Task tree) Logical regions Separation of algorithm from scheduling Mapping interface	Implicit scheduling Explicitly express data/task parallelism Hardware generalization Optimization

Table 3.3: Useful abstractions from non-CIM frameworks and the underlying meta-abstraction they expose to programmers.

Parallel computers, discussed in Section 2.3.2, and in particular their taxonomies proposed by Flynn and Duncan (Figures 2.15 and 2.16), had us considering to categorize CIM devices within a single class of parallel computers and, from there, identify the most appropriate programming model for that class. However, upon closer examination, it became evident that CIM accelerators cannot be confined to a single category. Different implementations exhibit characteristics that overlap multiple paradigms, including pipelined vector processors, systolic arrays, and dataflow machines. Even GPUs cannot be precisely classified within these taxonomies: while warps operate as SIMD units executing the same instruction, they also display associative processor-like behavior, as their computations can vary slightly based on local data. Furthermore, modern GPUs increasingly incorporate MIMD capabilities [65].

The CTA model could be used in a CIM programming model as a hardware abstraction. In this context, each processing element in the CTA can be mapped to a CIM macro. The strength of this approach lies in the flexibility of the CTA model with respect to network topology and communication mechanisms, making it well-suited to capture the complexity found in CIM. Moreover, the distinction between local and remote memory access latencies aligns closely with the physical characteristics of CIM macros, which operate on local data but incur higher latency when accessing data from other arrays. However, this view overlooks the hierarchical structure present in general CIM hardware abstraction layers, as illustrated in Figure 3.2 and reflected in OpenCL's platform model.

OpenCL, presented in Section 2.3.3, is the only framework in Table 3.3 that uses eager execution. In contrast, recent work is shifting toward deferred execution, making an OpenCL-style extension for CIM possible but not ideal for future-proofing. Still, OpenCL's parallelism abstractions could be reused in a deferred CIM model, and its hierarchical platform model—similar to the hardware abstraction in CIM frameworks (Figure 3.2)—could also be leveraged.

Halide, covered in Section 2.3.4, has already inspired the CIM domain, with the CIM-Explorer framework discussed in Section 3.1. Its abstractions—such as the separation of algorithm and scheduling—support the development of more effective compilers. Additionally, its declarative scheduling primitives could offer programmers a degree of control over the mapping process onto CIM macros. Following CIM-Explorer's approach, the best path forward is to extend the TVM [66] compiler, which—unlike Halide—is not limited to image processing and provides a more general foundation for CIM support.

Legion, described in Section 2.3.5, would be a great fit for CIM. Legion shares several similarities with Halide, including the separation of algorithm and scheduling, deferred execution, and implicit parallelism. In addition, Legion is extensible, capable of supporting different coprocessors by filling in

some predefined interfaces. While this is possible, it introduces unnecessary complexity as programmers who only aim to target CIM would be required to adopt Legion, a more general framework whose broad design may prevent it from fully exposing CIM-specific abstractions to the programmer.

Apache Spark's programming model, explored in Section 2.3.6, is incompatible with CIM, as it prioritizes resilience through its primary abstraction RDDs. In CIM, resilience does not need to be implemented in software, since faults are typically handled at the hardware level. However, Spark's shared variables could still be leveraged to introduce data types with predefined communication patterns that reduce data transfer overhead. This helps compilers produce more efficient implementations.

3.3. Essentials for CIM programming models

In this section, we address **SQ1**, introduced in Section 1.3, by identifying the meta-abstractions that the abstractions of a CIM programming model should implement. These meta-abstractions, along with several abstractions that embody them from Sections 3.1 and 3.2, are shown in Table 3.4 and are organized according to the four key qualities of programming models outlined in Section 2.3.1: productivity, portability, performance, and expressiveness.

Programming Model Qualities	Feature	Abstractions
Productivity	Implicit scheduling	Dataflow execution model Deferred execution model
Portability	Hardware generalization	Platform model Seperation of algorithm and scheduling
Performance	Optimization	Declerative scheduling directives Mapping Interface, Shared variable types
Expressiveness	Explicitly express parallelism	ND ranges, Command queues Forming DFG using dataflow constructs Logical regions

Table 3.4: Essential meta-abstractions of CIM programming models, categorized by programming model quality. For each feature, we identify the abstractions that can be used to realize it, based on the analysis in Sections 3.1 and 3.2.

Productivity in programming models refers to enabling programmers to efficiently develop CIM applications. The key question, then, is what aspects programmers should not need to manage directly, and should instead be handled automatically. From Sections 3.1 and 3.2, we observe that most existing programming models employ implicit scheduling, allowing programmers to specify what should be done, but not how it is executed. However, most CIM frameworks adopt a declarative or API-driven approach. This is largely due to the inherent complexity of CIM accelerators, which motivates delegating much of the scheduling responsibility to the compiler. Consequently, future CIM programming models should preserve implicit scheduling by adopting deferred or dataflow execution models.

Both deferred and dataflow execution models offer useful properties for CIM. Deferred execution employs lazy evaluation to construct a computation graph that can be optimized and later mapped to hardware. In contrast, dataflow execution is driven by data availability and aligns more closely with the runtime behavior of CIM hardware, explicitly expressing concepts such as pipelines and streams. Their primary difference lies in how they implement implicit scheduling. Deferred execution performs scheduling at compile time, whereas dataflow execution does so at runtime, providing greater flexibility to handle dynamic workloads. The choice between the two depends on the applications the programming model aims to support. A practical middle ground is to combine both approaches, using deferred execution to build the computation graph and dataflow semantics to execute it, as in Section 4.1.1.

Portability in programming models refers to the ability of programs to execute across different hardware platforms without requiring programmers to rewrite applications. Achieving portability requires abstractions that enable hardware generalization. As observed in Sections 3.1 and 3.2, this can be achieved by separating the algorithm from its scheduling or by adopting a platform model.

Separating the algorithm from scheduling divides the program into two parts. The first specifies the computation to be performed and can be reused across different hardware implementations. The second part defines or hints at scheduling decisions for the compiler to optimize execution on a specific architecture. Even with implicit scheduling, guiding the compiler can be highly valuable for optimization.

A CIM platform model should be sufficiently general to accommodate the complexity of diverse CIM implementations while highlighting the commonalities among them. As discussed in Section 2.1, different implementations may vary in their circuits, organizations, interconnections, and supported operations. Ultimately, the key commonality across all designs is the CIM primitive's 2D structure. Furthermore, as outlined in Section 3.1, several hierarchical hardware abstractions—such as the one illustrated in Figure 3.3—have been proposed in the literature to generalize across accelerators targeting specific domains, such as neural networks [45] or pattern matching [72]. We argue that this abstraction is sufficiently expressive to represent CIM accelerators regardless of their application domain. Although the terminology used for each tier may vary across the literature, we adopt the following naming scheme for each tier: CIM chip–CIM core–CIM macro. Inspired by the CTA model, aspects such as interconnection type, organization, peripheral circuitry, and the number of processing units are left intentionally ambiguous to maximize generality.

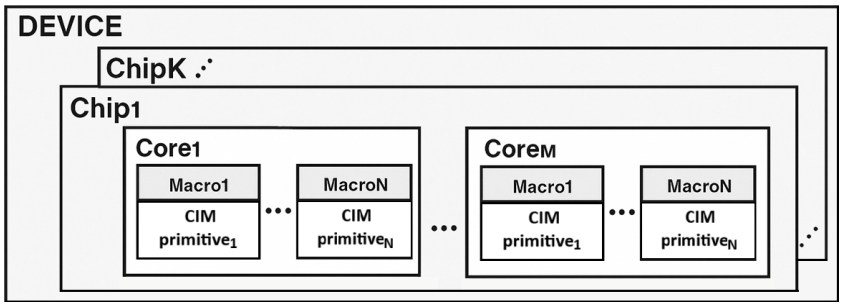


Figure 3.3: The four tier chip-core-macro-primitive hierarchical architecture that is popular for current homogeneous CIM devices. The names of each tier differs across the literature. Examples that use it include [45, 54, 55, 51, 46]

Performance in programming models refers to the ability to use high-level abstractions while achieving efficiency comparable to hand-optimized low-level code. Optimization encompasses not only performance but also the ability to balance additional metrics such as energy consumption and resource utilization. Based on our analysis, this can be achieved either through declarative scheduling primitives, as in Halide and CIM-Explorer, or through a more imperative approach, such as Legion's mapping interface—an object-oriented interface that designers implement to optimize performance. It is important to note that both approaches rely on the separation of algorithm and scheduling abstraction.

Shared variable types can also be used for optimization. Specifically, at the device level of CIM, as discussed in Section 2.1.1, current memristor implementations exhibit limited endurance and require high reprogramming energy. To make CIM a viable solution, frequent reprogramming operations should therefore be minimized. We believe that this critical issue should be addressed at the programming level. Different shared variable types could be associated with the memory technology on which they are stored. In cases where memristive memory is used, immutability constraints—similar to those found in functional programming, could prevent programmers from unintentionally writing energy-inefficient applications, albeit at the expense of reduced expressiveness. We implement such a feature in our proposed programming model in the form of different memory objects introduced in Section 4.1.3.

Expressiveness in programming models refers to the ability to implement a wide range of applications effectively. A CIM programming model should therefore provide programmers with the appropriate abstractions to use as tools to express a plethora of applications. Explicitly specifying parallelism helps expressiveness because it expands what computations a programmer can represent directly within the programming model. In addition, it makes it easier for the compiler to extract parallelism from programs. Various forms of abstractions can be used to achieve this such as ND-ranges and Command queues that split the responsibility of data and task parallelism respectively. Other options include using dataflow constructs such as nodes, streams and pipelines that operate on multidimensional data. In Section 4.1.5, we introduce a CIM data-parallel abstraction inspired by Halide's tiling.

In conclusion, we recommend that CIM programming models implement abstractions that realize the following four meta-abstractions: implicit scheduling, hardware generalization, optimization, and explicit parallelism. For each feature, we have identified existing abstractions that can be adapted for use in CIM. Nevertheless, new abstractions specifically designed for CIM accelerators may offer improved suitability. To demonstrate the effectiveness of these concepts, we propose our own programming model in Chapter 4 that includes both reused and novel abstractions.

4

Proposed programming model

This chapter presents our main contribution a general purpose programming model for CIM accelerators. Initially, building on our definition of a programming model we define the underlying computer model and the programming abstractions we have chosen. Subsequently, we demonstrate the expressiveness of our programming model by implementing three CIM applications, providing both code and motivational examples to illustrate them.

4.1. Programming model specification

In Chapter 3 we propose that CIM programming models need to contain abstractions that enable the following meta-abstractions: implicit scheduling, hardware generalization, optimization and explicit expression of parallelism. Table 4.1 shows the list abstractions we have chosen to implement these meta-abstractions. The following sections go into the details of each abstraction in order from top to bottom. At each section we will incrementally build a DSL using our programming model introducing the syntax and semantics along the way.

Programming Model Qualities	Related meta-abstraction	Abstractions
Productivity	Implicit scheduling	Deferred execution model
Portability	Hardware generalization	Platform model Separation of algorithm and scheduling
Performance	Optimization	Shared variable types (Memory model) Declarative scheduling directives
Expressiveness	Explicitly express parallelism	Tiling Forming DFG using kernels

Table 4.1: Essential meta-abstractions of CIM programming models, categorized by programming model qualities. For each meta-abstraction, the corresponding abstractions used to realize it in our programming model are listed.

4.1.1. Deferred execution model

The execution model defines how instructions are scheduled and processed on the platform. For simplicity, and because it suffices to understand the execution model, we assume a platform consisting of a general-purpose host connected to a CIM accelerator. We employ a deferred execution model, meaning that operations are not executed immediately but are instead recorded in an acyclic dataflow graph (ADG) which is scheduled at compile time. This graph represents a multi-stage pipeline executed with dataflow semantics, as shown in Figure 4.1a. Each node in the ADG corresponds to a logical CIM macro that applies a specific set of operations to an incoming data stream and produces an output stream. The host constructs this ADG incrementally during program definition. Once the graph is fully defined, it is optimized and offloaded to the CIM accelerator for execution through a dedicated function call, `realize()`, which triggers its evaluation on the device.

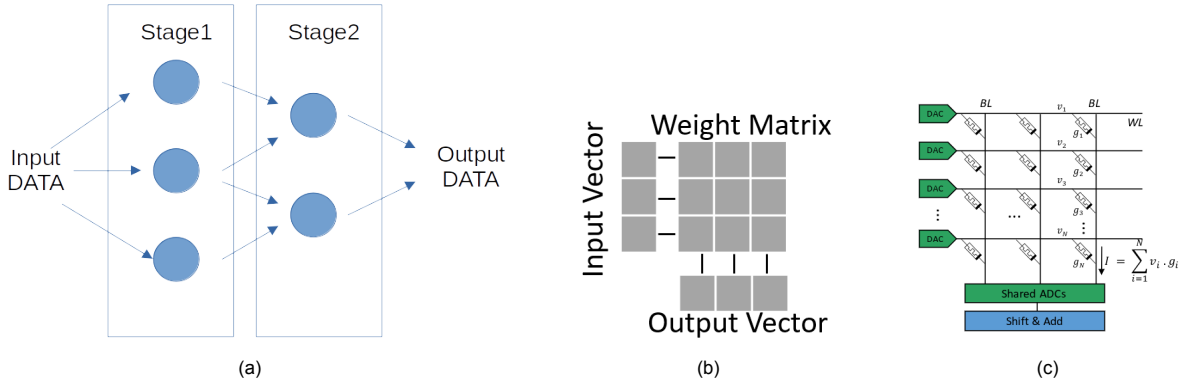


Figure 4.1: (a) A two-stage pipeline defined as an ADG. Each node represents a logical CIM macro that processes a stream of data. Nodes within a pipeline stage perform the same operation on different data points. (b) Logical CIM primitive components illustrating input/output vectors and a weight matrix. (c) Physical CIM primitive (CIM-Xbar of binary memristive cells) [49].

Before proceeding, it is important to distinguish between logical and physical CIM primitives or CIM macros as illustrated in Figure 4.1b and Figure 4.1c, respectively. Logical units represent abstract computational units expressed in a program, describing how data and operations are conceptually organized within the system. Physical units, on the other hand, correspond to the actual hardware arrays composed of memristive cells that perform the computations. The programming model focuses on the logical level and not how the computation will be mapped to hardware. Hence, we ignore mapping details such as converting between memory cell precisions and different memory array dimensions. We discuss mapping in more detail in Section 5.5.

Our execution model adopts a kernel-parallelism design pattern similar to OpenCL, as discussed in Section 2.3.3. A kernel is a programmer-defined function representing a single pipeline stage in the ADG, and one kernel instance executes on each logical CIM macro in that stage. As shown in ADGs like Figure 4.1a, data parallelism arises by grouping CIM macros within the same stage, where each performs the same operation on different data elements. In contrast, task-level parallelism appears across pipeline stages as data flows from one CIM macro to the next.

Similar to OpenCL, to maximize performance, kernels must be mapped efficiently onto physical CIM macros. Each kernel should exploit the full height and width of the CIM primitive it is assigned to in order to maximize parallelism per processing unit. In addition, data should be distributed evenly across processing units to avoid load imbalance. The ADG itself consists of logical CIM macros that execute concurrently, provided sufficient resources are available. If resources are limited, one or more nodes must execute sequentially, which degrades performance.

4.1.2. Platform model

The platform model is essential because CIM accelerators can differ at every level of design, from the circuit primitives to the overall system organization. Although our goal, as mentioned in Section 1.4, is not to develop a programming model that unifies all these diverse architectures, we must still represent the underlying machine through an abstraction that captures its key characteristics while remaining flexible enough to accommodate variations. Hence, machines that adhere to this platform model are considered compatible with our programming model. Moreover, the platform model not only hides the complexity specific to each accelerator but also highlights their shared features for the programmer.

Our platform model, shown in Figure 4.2, consists of a general-purpose host and a single CIM device that extends the homogeneous hierarchical hardware abstraction introduced in Section 3.3. Each CIM macro contains a single CIM primitive, but we make no assumptions about its dimensions or supported operations. At every level of the hierarchy, the model includes a memory region or buffer and a conventional post-processing functional unit (FU), similar to CIM-MLC's hardware abstraction [45]. Following the CTA model in Section 2.3.2, we also avoid assuming any specific interconnect topology, as such details are complex, low-level, and best left to hardware designers rather than programmers.

The design choices behind the platform model were made to maximize applicability across a variety of implementations. The ambiguity of the exact specs of the CIM macro and post processing circuits allow the coexistence of implementations with different supported operations. Furthermore, the hierarchy

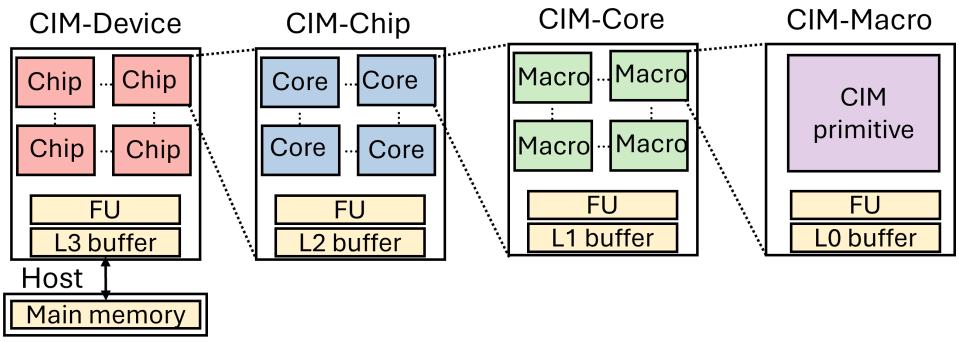


Figure 4.2: Our platform model consisting of a homogeneous hierarchical CIM accelerator and a general-purpose host.

and lack of interconnection specification are flexible enough to accommodate different organizations beyond just a heterogeneous system. Specifically, by assuming that each tier in the hierarchy contains a single element, we can represent single CIM macro accelerators [74] . Similarly, by assuming a single CIM chip and CIM core, we can express a dataflow architecture between CIM macros. Furthermore, the hierarchical structure can represent a distributed system, where the groupings reflect the relative proximity among CIM macros. Similar to the CTA, Figure 4.2 does not assume a memory reference mechanism such as shared memory or message passing.

4.1.3. Memory model

The memory model defines the system's disjoint memory regions and the corresponding dataflow of execution. As illustrated in Figure 4.2, we distinguish between two types of memory: the memristive memory that constitutes the CIM primitives, and the conventional memory present at each level of the hierarchy. As discussed in Section 2.1.1, reprogramming memristive cells is costly in terms of both energy consumption and cell endurance. Consequently, conventional memory is employed to store intermediate results that are frequently overwritten. Depending on the application, this conventional memory may serve as input/output buffers within the CIM macro, or as scratchpad memory.

The hierarchy, shown in Figure 4.2, highlights the different communication latencies that exist between processing units. Two CIM macros inside the same CIM core can exchange data faster than CIM macros in different CIM cores, and the same idea holds when comparing communication within a CIM chip versus across CIM chips. Therefore, programmers should try to make the most of spatial locality at each level of the hierarchy to reduce communication overhead.

Within the memory model, two types of memory objects are defined: Tensors and Streams. These abstractions represent distinct ways of organizing data. Tensors represent multi-dimensional data layouts, while Streams describe the data movement between different processing stages.

A Tensor, illustrated in Figure 4.3a, is an ordered, immutable, multidimensional array that contains elements of a single data type, with all values fully computed and available. A Tensor is considered ordered because its elements and dimensions follow a defined sequence that determines how data is laid out and accessed in memory. Tensors are intended to be distributed across the various CIM macros of the accelerator and stored within the weights of the corresponding CIM primitive.

A Stream, shown in Figure 4.3b, is a handle to an ordered, 1D sequence of elements of a single data type. The data in a Stream may not be immediately available and can be of unbounded length. Unlike Tensors, Streams are designed to feed preloaded CIM macros by continuously transferring data through their input vectors as illustrated in Figure 4.3c. Streams also feature a sliding window, enabling the processing of multiple elements in batches. The default stride for the window is one element.

Figure 4.3c illustrates an abstract example of the system's dataflow. First, Tensors are mapped onto CIM primitives. For simplicity, we only show two CIM primitives; however, the actual number depends on the size of the `x,y,z` dimensions. Data elements are then streamed from the input stream and broadcast to all CIM primitives. Once the results are computed (shown in purple), they are merged into a new output stream. This provides an abstract representation of this type of computation, where the exact configuration of the Tensor, the window size of the Stream, and the reduction depend on the specific operation being performed.

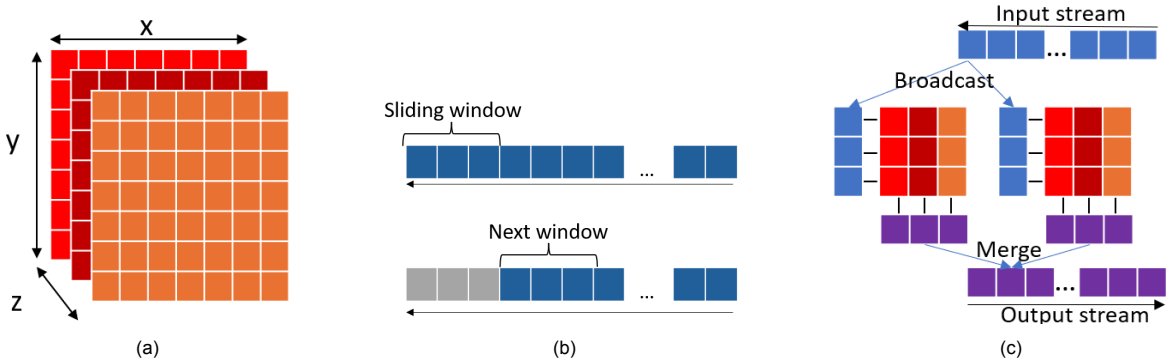


Figure 4.3: (a) A 3D Tensor with dimensions `x`, `y` and `z` organized in this order. (b) Illustration of a Stream flowing from right to left with a sliding window and stride of three elements. (c) Illustration of an input stream feeding two preloaded logical CIM primitive units, with intermediate results merged to create an output stream.

4.1.4. Separation of algorithm and scheduling abstractions

The separation of algorithm and scheduling, as shown in Chapter 3, is a prevalent feature of many programming models, both CIM and non-CIM. We therefore considered it a mandatory design aspect. The key question then becomes how to realize this separation. In our model, this is achieved through three abstractions: dimension-agnostic kernels, symbolic variables, and declarative scheduling directives.

We propose dimension-agnostic kernels as a novel abstraction that extends the kernel design pattern by decoupling the algorithm from scheduling, as illustrated in Listing 4.1. Similar to Halide [48], the `Func` output type represents intermediate results. Its interpretation is either a Tensor or Stream depending on the type of the function parameter it is passed to. Tensors use angle brackets `(<>)` to specify the type of their internal elements and the sizes of their dimensions, while Streams use them to specify the element type and the size of the sliding window and the stride. All sizes are expressed using the symbolic variable type that provides generality. Moreover, Streams enable implicit parallelism throughout the kernel's execution for each chunk of data derived by applying the sliding window to the stream, whereas tensors persist across iterations. This design choice encourages writing programs that avoid modifying the CIM primitive.

An immediate question is what operations are available within a kernel. In our platform model definition, we make no assumptions about the supported operations in order to remain as general as possible. Consequently, we assume that the underlying accelerator can perform any basic micro operation that can be implemented on CIM, such as MVM or search operations. In our examples, in Section 4.2 we assume that all CIM operations are contained in the `cimOps` object, and that all peripheral operations are contained in the `aluOps` object that are assessable inside all kernels. We discuss what operations should be available within a kernel in Section 5.3.

Symbolic variables in kernels express relationships between input and output dimensions that hold independently of their concrete values. In Listing 4.1, the output's size (`N`) matches the outer dimension of `weights`, while its inner dimension corresponds to the sliding-window size (`M`) of `s0`. More complicated relationships can also be formed such as using operators to an output dimension that is the product of two input dimensions. The combination of symbolic variables and dimension-agnostic kernels lay the groundwork for the data-parallel tiling abstraction in Section 4.1.5.

Similar to Halide, our schedule directives are declarative loop transformations that take symbolic variables and integers as arguments. We adopt this approach to allow programmers to influence scheduling for optimization purposes. In our model, scheduling directives are expressed as method calls applied to kernel invocations. The complete list of scheduling primitives used in this paper is provided in Appendix A following Halide's notation.

```
1 Func<int64,N> Kernel(Tensor<bit, N, M> weights, Stream<bit, M, stride> s0) {
2     return cimOps.mvm(weights,s0);
3 }
```

Listing 4.1: Example kernel definition accepting a 2D tensor and a stream of sliding window size M that perfroms MVM.

4.1.5. Expressing Data-Parallelism using the tiling abstraction

Data parallelism can be defined as performing a set of operations on multiple data points in parallel. To efficiently exploit data-parallelism in CIM, compilers must easily derive data mapping from source code. The abstractions listed in this section aim to make data parallelism more explicit.

To express data-level parallelism we were inspired by the `tile` scheduling primitives used by Halide's programming model, as explained in Section 2.3.4. Pelke et al. [56] have already adopted tiling for programming both Binary and Ternary Neural Networks in CIM. Our contribution is to generalize this idea for programming other applications outside the machine learning domain and to extend tiling to represent both data parallelism, distribution and reduction.

In Figure 4.4, we provide a conceptual example of how tiling can be applied in CIM. We assume that our input is a 1D Tensor of data shown in Figure 4.4a. Our goal is to distribute this Tensor efficiently across the accelerator's CIM macros, as weights. By using tiling, we can change the dimensionality of the input data while preserving order, assigning the new dimensions to useful concepts using Halide's symbolic variable type to represent iteration. For example, in Figure 4.4b, we introduce the `dimX` symbolic variable that we bind to the width of the CIM primitive. Then the input data is rearranged into a 2D Tensor that can be directly mapped to a CIM macro with a sufficient height. However, depending on the architecture, such a large CIM macro may not be available. In that case, as shown in Figure 4.4c, the data can be split into a 3D structure, where each 2D slice is distributed to a different CIM macro. Yet again, the system may not have a sufficient number of CIM macros to execute all operations in parallel. In Figure 4.4d, we introduce another dimension representing time. This example demonstrates how Halide's tiling naturally synergizes with the 2D structure that CIM primitives of all CIM accelerators have in common allowing programmers to influence scheduling.

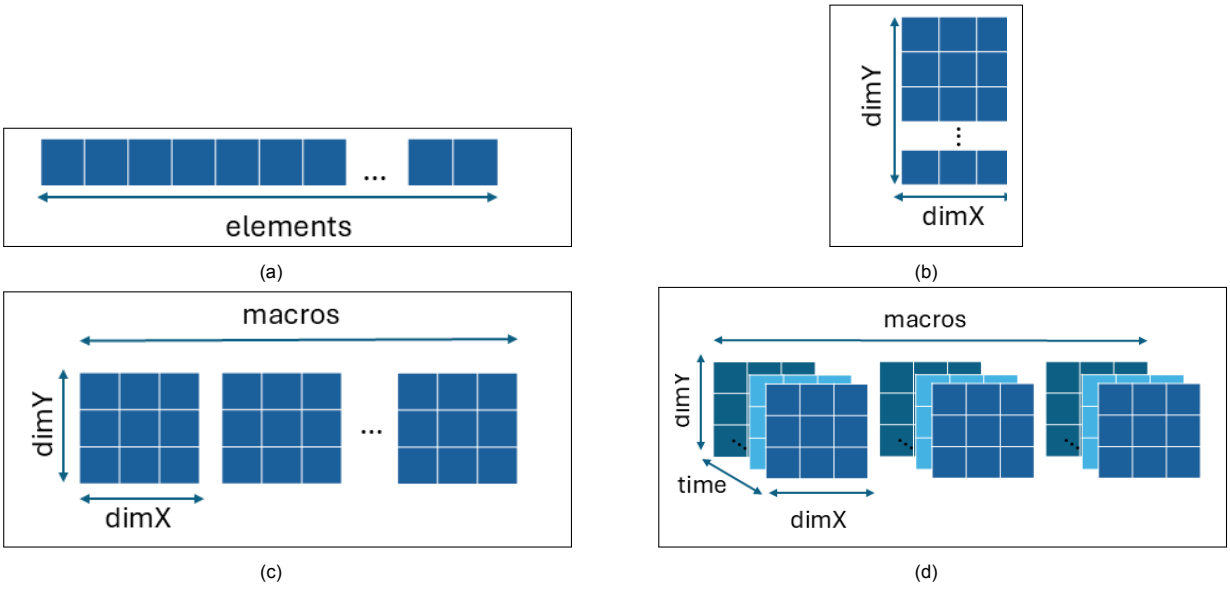


Figure 4.4: (a) 1D Tensor before tiling. (b) 2D Tensor that can be mapped to a sufficiently large CIM macro. (c) 3D Tensor that can be mapped to multiple 3x3 CIM macros. (d) 4D Tensor that allows multiple weights to be applied to many 3x3 CIM macros over time.

An abstract tiling example is shown in Listing 4.2. Initially, we define a set of scalar values on the host: the data width, number of input data elements and a stride. Next, we declare the required Tensors and Streams on the host. In the angle brackets (<>), both specify an element type, but Tensors define their structure through dimensions, whereas Streams additionally specify a sliding-window size and a stride. We then offload the Tensors to the accelerator. Note the use of symbolic variable instances `x` and `y`, which are bound to the dimensions of the `weights_on_acc` tensor. Afterward, we invoke the kernel defined in Listing 4.1. Finally, we use the symbolic variables as arguments to scheduling primitives that guide the program's mapping.

In Listing 4.2, we use the `tile` and `bind` scheduling primitives. First, `tile` is used to split the `weights_on_acc` matrix—whose dimensions are bound to `y` and `x`—into the internal dimensions `yi` and `xi`. These have sizes `M_HEIGHT` and `M_WIDTH`, respectively, which—together with `CORES` and

MACROS—are compile-time constants defining the dimensions of the CIM primitives and the number of CIM macros and CIM cores in the platform defined in Section 4.1.2. Then, we use bind to assign each dimension to a tier in the hierarchical platform model. The resulting schedule interpretation is shown in Listing 4.3 where parallel for loops are introduced to represent parallelism across CIM core and CIM macro units. At the same time, at each call, a matrix of data is fed into the function call based on the dimensions of the macro. For brevity we omitted the indexing of the result type. The nested loop is repeated sequentially for each sliding_window in the s0 stream.

tile and bind are just two possible scheduling primitives from Table A.1. Dimensions can be assigned to other concepts as well, explored more in Section 4.2. The result is an abstraction that is capable of expressing much more than just data parallelism. For example, the sequential and reorder directives can be used to define sequential iteration in the case of insufficient resources, something that OpenCL's ND-ranges cannot do.

```
int64 n_elem, data_width, stride = 10_000, 32, 32;
Tensor<bit, n_elem, data_width> weights_on_host = get_weights();
Stream<bit, data_width, stride> s0 = get_stream();
Func weights_on_acc = weights_on_host;
Func result = Kernel(weights_on_acc(y,x), s0);
      .tile(y, x, yi, xi, M_HEIGHT, M_WIDTH)
      .bind(y, CORES)
      .bind(x, MACROS)
      .bind(yi, M_HEIGHT)
      .bind(xi, M_WIDTH);
```

Listing 4.2: Abstract example of a call to the kernel defined in Listing 4.1. The symbolic variables highlighted in red are bound to the dimensions of the weights_on_acc functor.

```
for sliding_window in s0:
  parallel(cores) for i_y in y:
    parallel(macros) for i_x in x:
      result[...] = Kernel1(weights_on_acc(
                                  i_y*macro_ydim:macro_ydim+i_y*macro_ydim,
                                  i_x*macro_xdim:macro_xdim+i_x*macro_xdim
                                  ),
                            sliding_window
                            );
```

Listing 4.3: Interpretation of the scheduling directives applied on Kernel in Listing 4.2. To sumarize, tiling is used to introduce a nested for loop. Each for loop is assigned a resource core, macro to be parallelized.

4.1.6. Expressing Task-Parallelism using kernel defined DFG

Task parallelism refers to executing multiple distinct tasks in parallel, each operating on the same data point in a pipeline fashion to collectively produce a final result. In programming, this is typically represented by deriving an ADG from the source code, which is subsequently mapped to hardware. The abstractions introduced in this section are intended to facilitate the extraction of task-parallelism.

Initially, we turned to Halide's programming model, which implements the pipeline design pattern [78, p.103] for constructing image processing pipelines. This dataflow coordination is made explicit using asynchronous programming to construct an ADG using functors (Func) that are produced from kernel invocations. Dependencies are constructed by passing the resulting functors as input arguments to subsequent kernels. However, by default, these pipelines execute sequentially, with stages interleaved to maximize locality. Although Halide provides the asynch directive to enable asynchronous pipeline execution, it is rarely used due to the difficulty of achieving effective load balancing. However, through our implementation of data parallelism in Listing 4.2, we provide users with greater control over load balancing, thereby addressing this issue.

In contrast to Halide, in our programming model a single kernel corresponds to a single stage in a data-flow pipeline. Consequently, all pipeline stages can execute in parallel, provided sufficient resources are available. Similar to Halide, the parameters passed into a kernel serve as pipeline inputs, while output functors can be consumed by other kernels, explicitly defining data dependencies within the program.

An example of a pipeline consisting of a single linear chain of stages—referred to as a linear pipeline—is shown in Listing 4.4 and illustrated in Figure 4.5a. The example program defines four pipeline stages, each with a single Tensor argument. The first stage additionally receives a Stream as input, while the remaining stages consume the output functor of the preceding stage. For this example, we assume that the second parameter of all kernels is a stream, so all functors are treated as streams.

We employ asynchronous execution semantics so that no computation is offloaded to the accelerator until the `realize` function is invoked. The `realize` function is a blocking call that accepts a handle to a CIM device (in case multiple devices are connected to the host). Execution resumes only after `stage4` has been computed and offloaded back to the host.

```
/// ...
/// Define tensors and streams before this point
/// Define Pipeline
Func stage1 = Kernel1(tensor1, stream1);
Func stage2 = Kernel2(tensor2, stage1);
Func stage3 = Kernel3(tensor3, stage2);
Func stage4 = Kernel4(tensor4, stage3);
/// Execute pipeline (blocks)
Tensor<...> host_location = stage4.realize(device1);
```

Listing 4.4: Example of a four stage pipeline constructed asynchronously. All pipeline stages are preloaded with tensors then streams are fed into the resulting pipeline, forming explicit dependencies between stages. The resulting ADG is only executed once `realize` is called.

Figure 4.5: Example pipelines illustrations from [78, p.106]. (a) Linear pipeline: sequential chain of stages. (b) Nonlinear pipeline: data may be split to form parallel stages that perform the same operation. Introduces data parallelism for load balancing.

Resource allocation for each pipeline stage is handled by introducing data parallelism: symbolic variables can be used on pipeline input tensors as explained in Section 4.1.5, to distribute the workload across multiple CIM macro units, enabling fine-grained control over the resources assigned to each stage. This allows us to construct nonlinear pipelines, as illustrated in Figure 4.5.

4.2. Implemented applications

Now that we have a basic understanding of our programming system, we evaluate it by expressing the CIM applications introduced in Section 2.2. For each CIM application, we provide the code snippet that describes the computation along with the kernels it relies on. We assume access to a programmable accelerator that conforms to the computer model defined in Sections 4.1.1 to 4.1.3 and consists of a single chip with two cores, each containing two macros. In addition, we assume that the accelerator provides the CIM primitives required to support the operations used in each CIM application.

4.2.1. Integer Sorting

We implement an example of integer sorting assuming the existence of a programmable accelerator with an architecture similar to MemSort, as described in Section 2.2.1. The example considers a problem size of eight integers—chosen for illustration purposes, though it can be easily extended to larger datasets. Since the accelerator consists of four CIM macros, and each CIM macro can store up to two integers, all eight integers fit on the accelerator at once.

The program designed for the accelerator can be represented as a three-stage pipeline, shown in Figure 4.6, which processes one integer at a time to determine its exact position in the sorted array. In the first stage, the input integer—71 in this example—is compared against all other integers in the unsorted stream, producing a binary array where a one indicates that the input is less than the corresponding element. This includes the CIM macro that contains the integer as it needs to placed at

its input buffer. In the second stage, a popcount operation determines the index position, and in the third stage, the element is placed at the calculated position. Figure 4.7 illustrates the same process as Figure 4.6 but on a simplified illustration of the assumed hardware.

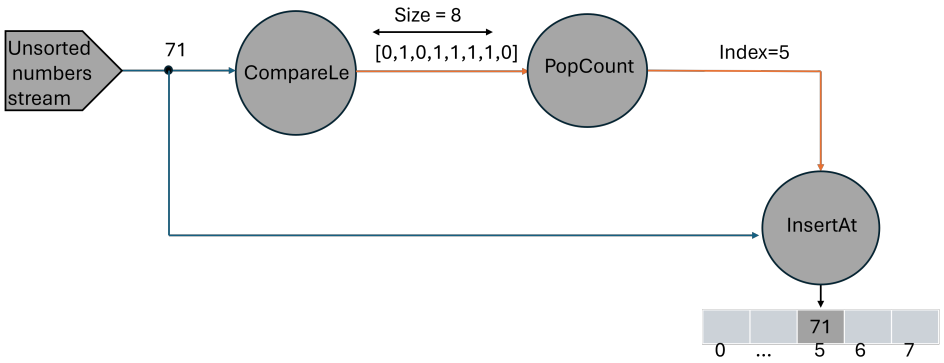


Figure 4.6: Dataflow graph for the sorting application, consisting of three stages: comparing the input integer (71) with all others, computing its index via popcount, and placing it in its sorted position.

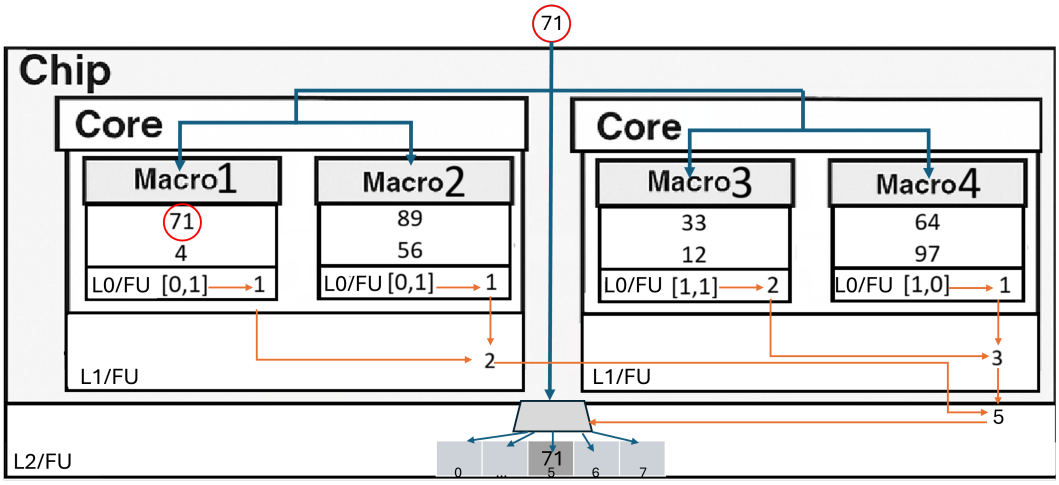


Figure 4.7: Illustration of a single iteration of sorting on an accelerator with one chip containing two cores, each with two macros. Arrow colors match Figure 4.6. The input integer (71) is broadcast (blue) from `Macro1` to all other macros, then a reduction (orange) determines it belongs at index 5.

The code example for the sorting application is shown in Listing 4.5. It begins by defining a one-dimensional Tensor of unsorted integers on the host. Subsequently, by placing it on the right-hand side (RHS) of a functor assignment, the unsorted integers are offloaded from the host to the accelerator. From this functor, we then define a handle to a stream. As a result, the unsorted integers are first stored on the CIM primitive units and later streamed through the pipeline once it is constructed. Next, we define the pipeline stages. The `CompareLe` kernel, shown in Listing 4.6 performs element-wise comparisons between a single element—referred to as the streamed integer—and the entire tensor of unsorted integers, returning a functor of the same length whose entries are binary values indicating whether the streamed integer is less than or equal to each CIM array entry. The resulting functor is then passed as a tensor to the `PopCount` kernel, where it is aggregated into a single value representing an index. Finally, the streamed element is inserted into the sorted array.

We now focus on the scheduling portion of Listing 4.5. Using the `split` directive, we change the dimensionality of the unsorted tensor from that in Figure 4.8a to that in Figure 4.8b. Next, with the `bind` directive, we explicitly specify how the unsorted integers are laid out on the accelerator to produce the result shown in Figure 4.7. As a consequence, the interpretation of the schedule transitions from the sequential version in Listing 4.7 to the parallel version in Listing 4.8, where comparisons are parallelized both within CIM macros and across CIM macros and CIM cores.

Apart from `split` and `bind`, we use two additional scheduling directives. With `reduction_order`, we specify that the summation occurring in the `PopCount` kernel must take place first within CIM macros and then across CIM macros and then CIM cores. In addition, we indicate that the `sorted` list must be stored in L2 memory using `malloc_at`.

By calling `realize` the program blocks until the execution on the accelerator finishes. While we have specified that the allocation size of the resulting array this value can also be derived from the kernel signatures used in the computation and their input parameters at compile time. This allows the compiler to verify memory correctness and avoid unnecessary over-allocation.

```
/// Definitions of data stored on host
int64 n_elem = 8;
Tensor<int64, n_elem> unsorted_on_host = read_file();

/// Algorithm defines pipeline
Func unsorted_on_acc = unsorted_on_host(x);
Stream<int64, 1, 1> unsorted_stream = unsorted_on_acc;
Func comparisons = CompareLe(unsorted_on_acc, unsorted_stream);
Func index = PopCount(comparisons);
Func sorted[index] = unsorted_stream;

/// Schedule
unsorted_on_acc
  .split(x, i_cores, i_macros, i_m_height, _,  _, M_HEIGHT)
  .bind(i_cores, CORES)
  .bind(i_macros, MACROS)
  .bind(i_m_height, M_HEIGHT)
index
  .reduction_order(i_m_height, i_macros, i_cores)
sorted
  .malloc_at(L2)

/// Execute. Blocks until completion
Tensor<int64, n_elem> result = sorted.realize()
```

Listing 4.5: Example of sorting implemented using our programming model. The symbolic variable `x` is red to highlight that it is bound to the dimensions of the `unsorted_on_host` tensor. All other symbolic variables begin with the letter i to avoid confusion with constants, which are written in uppercase and may have similar names.

```
Func<int64,N> CompareLe(Tensor<int64, N> unsorted, Stream<int64, 1, 1> s0)
{
    return cimOps.compare(unsorted, s0, LE);
}
Func<int64,1> PopCount(Tensor<int64, N> comparisons)
{
    return aluOps.reduce_sum(comparisons);
}
```

Listing 4.6: Kernel definitions used in the sorting example in Listing 4.5.

```
for element in unsorted_stream:
  for i_x in x:
    comparisons[...] = CompareLe(unsorted_on_acc(x),element);
```

Listing 4.7: Interpretation of calling `CompareLe` in Listing 4.5 without scheduling applied. For each stream element (unsorted integer), comparisons with the unsorted integers stored in the CIM macros occur sequentially. Indexing is omitted for simplicity.

```
for element in unsorted_stream:
  parallel(CORES) for i_cores in CORES:
    parallel(MACROS) for i_macros in MACROS:
        parallel(M_HEIGHT) for i_m_height in M_HEIGHT:
            comparisons[...] = CompareLe(unsorted_on_acc(...),element);
```

Listing 4.8: Interpretation of calling `CompareLe` in Listing 4.5 with the `bind` and `split` scheduling directives. For each stream element (unsorted integer), comparisons with the unsorted integers stored in the CIM macros occur in parallel across CIM cores, across CIM macros, and within each CIM macro. Indexing is omitted for simplicity.

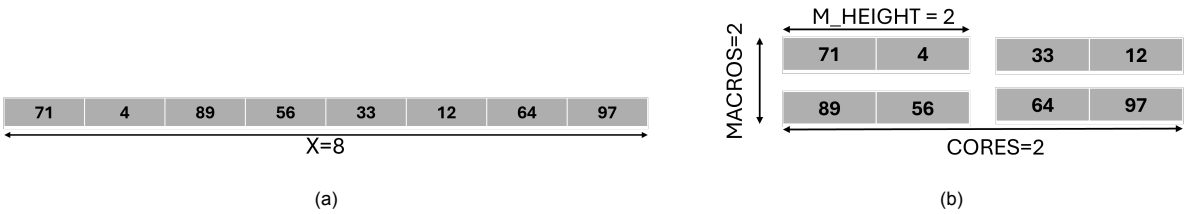


Figure 4.8: Dimensionality of the `unsorted_on_acc` tensor in Listing 4.5 (a) Without applying the `split` scheduling directive (b) With the `split` scheduling directive where the dimension order from innermost to outermost is `M_HEIGHT`, `MACROS`, `CORES`.

4.2.2. Pattern matching

In Section 2.2.2 we detail different types of possible match types. An example, of an exact match application is monitoring a network data stream to identify malicious patterns encoded in a lookup table (LUT). Once such a pattern is detected, we want to print the stream offset and the address of the matching case. Furthermore, in case of multiple matches we want to print the pattern with the lowest address on the host. In contrast with other application, the LUT is encoded in ternary values (trits).

The two-stage pipeline for this application is illustrated in Figure 4.9. In the first stage, a single bit is streamed into a shift register, which is then compared against the entire LUT. This comparison may produce multiple partial matches. For the notation of partial matches, we use the format Mx,ry, where x denotes the CIM macro number and y the row number within that CIM macro. Partial matches must then be gathered and resolved into a single LUT index in the second pipeline stage. Simultaneously, a per-bit counter is maintained in the first stage to track the current offset in the stream. Offset and match information are forwarded to the second stage only if at least one partial match is detected. The program should terminate after the second stage has produced its output.

Figure 4.10a shows the LUT used in our motivational example. It consists of four rows, with each row representing an entry in the LUT, colored distinctly. As discussed in Section 2.2.2, a key challenge in pattern matching applications is deciding how to distribute the data across the available memory arrays. Assuming we have only CIM-CAM units that can each store two rows of eight bits, there are two possible scheduling strategies. Figure 4.10b proposes a row-wise placement of LUT segments, while Figure 4.10c changes the order to a column-wise order. Each approach has different trade-offs, which we discuss below but both are expressible within our programming model.

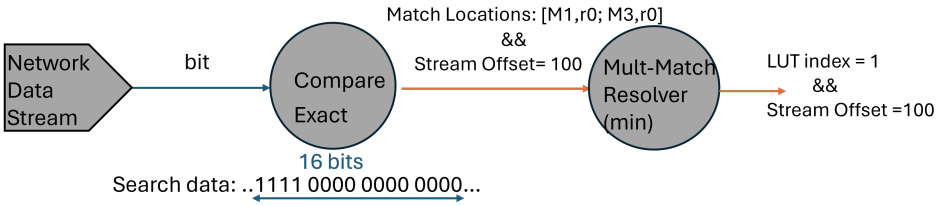


Figure 4.9: Dataflow graph for the pattern matching application. Blue lines represent data broadcast to the pipeline, while orange lines indicate reduction operations.

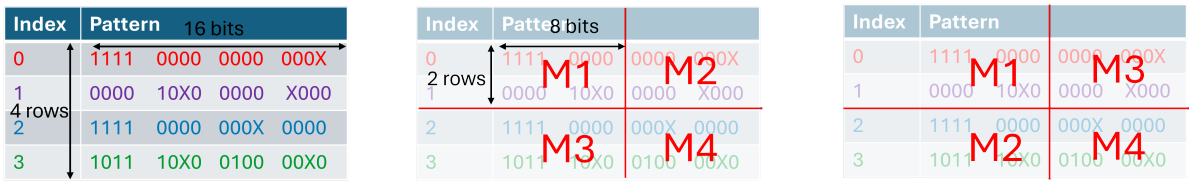


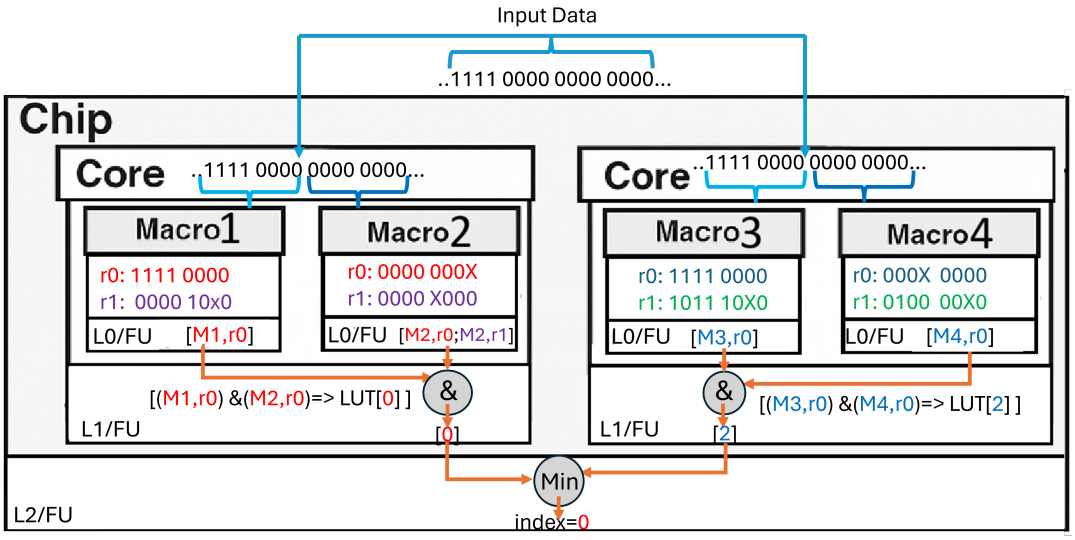
Figure 4.10: Different ways to split the 4×16-bit LUT among CIM macros of size 2×8-bit. (a) shows the original LUT with color-coded entries, (b) illustrates a row-wise distribution of segments, and (c) presents a column-wise distribution.

Assuming we choose the row-wise distribution shown in Figure 4.10b, the resulting execution is illustrated in Figure 4.11a. From this execution, we observe that the input data is broadcast to each

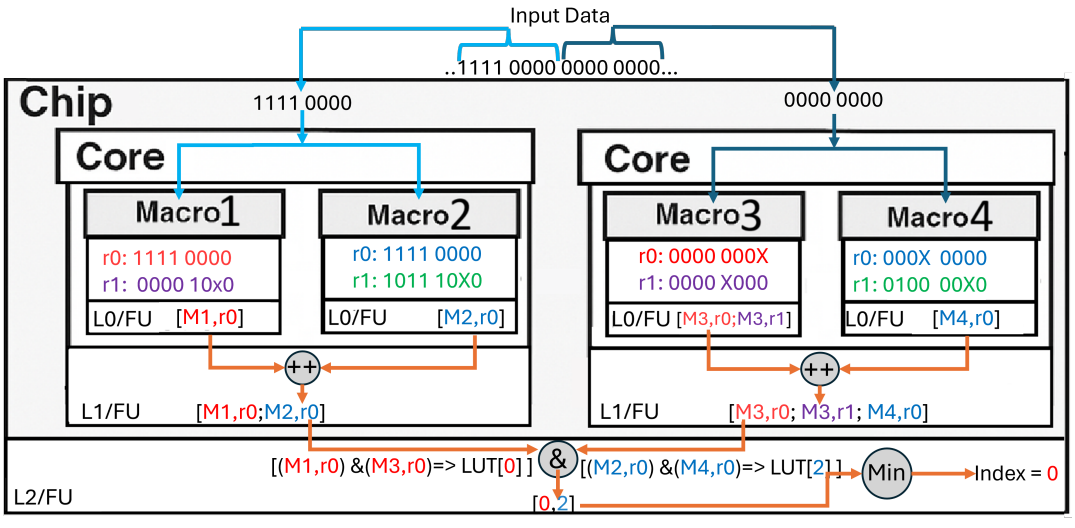
CIM core, and each CIM macro within a CIM core processes half of the input data. CIM macros within the same CIM core store an entire LUT entry and each CIM macro returns the addresses where matches are detected. Then these results are reduced using an AND (&) operation to determine which LUT row, if any, constitutes a complete match. Finally, a minimum operation is performed at the chip level to resolve cases where multiple matches occur.

On the other hand, the column-wise distribution shown in Figure 4.10c results in the scheduling illustrated in Figure 4.11b. In this case, each CIM core receives only half of the shift register reducing the input size required per CIM core. However, it makes the reduction phase more complex. Since individual LUT entries are distributed across multiple CIM cores, their partial results must first be gathered (++) at the chip level before the true matching indices can be determined. Hence, the column-wise case involves more post-processing data movement at the chip level however it could be favorable when CIM cores lack local post-processing units, either to reduce area cost or simplify the design.

Determining which of the two distributions is better depends on the underlying hardware. Regardless, our goal is to provide sufficient expressiveness in the programming model. This ensures that design space exploration is straightforward and that different schedules are easily expressible.



(a)



(b)

Figure 4.11: Different schedules of the pattern matching application. LUT entries in the CIM macros are from Figure 4.10, and the blue and orange lines are from Figure 4.9. Partial addressing uses Mxry (x = CIM macro, y = row). (a) Row-wise schedule, (b) Column-wise schedule.

Listing 4.9 shows the pattern matching program's code. The program begins by defining the required scalars, tensors, and streams on the host. This includes a 2D Tensor representing the LUT and the `data_stream` of bits that we monitor using a sliding window of length equal to the `pattern_size`. By default, all streams advance their sliding window with a step size of a single data element hence we omit the stride. Next, we offload the `lut_on_host` tensor to the accelerator. The algorithm is expressed as a two-stage pipeline, illustrated in Figure 4.9. In the first stage, `CompareExact` compares a window of the network stream against the entire LUT to identify matches and returns a tuple. The first tuple element contains the complete set of matching LUT addresses, organized as a 1D structure, while the second element represents the corresponding stream offsets. Both values are passed to the second stage, `MultMatchResolver`, which performs aggregation using a minimum operation.

Looking at the scheduling, we use the `tile` and `bind` directives to split the LUT both horizontally and vertically and distribute it across the accelerator's CIM macros. The resulting change in dimensionality is shown in Figure 4.12. Specifically, the original two dimensions are expanded into four to match the structure of the CIM macros. This transformation is also reflected in the interpretation of `CompareExact`, which in Listing 4.11 is a sequential operation comparing each entry to the stream's window size, whereas in Listing 4.12 it expresses parallel pattern matching across CIM macros, CIM cores, and within each CIM macro.

In addition, by using the commented-out `reorder` directive, we can switch from the row-wise schedule shown in Figure 4.11a to the column-wise schedule shown in Figure 4.11b. This demonstrates that we can switch between different schedules without modifying the core computation. Thus, programmers can potentially explore alternative mappings with minimal effort.

The kernel definitions used in Listing 4.9 are shown in Listing 4.10. The `CompareExact` function performs a `search` operation between the `lut` and `s0`, returning a list of matches of size `M`. We do this because, in reality, the number of matches cannot be determined exactly and is only guaranteed to be less than or equal to the number of LUT entries (`M`). By specifying the maximum output size explicitly, the programming model can allocate sufficient space during compilation while still allowing the number of matches to vary at runtime. The function also uses the `offset` operation, which returns an integer indicating the current position in the stream. Both values are returned as elements of a tuple, where each element is annotated using the `(type, length)` notation. The `MatchResolver` function then receives these matches, which may also be zero—in which case the kernel does not execute—and returns both the offset and the final resolved match as tuple elements.

Similar to the previous examples, once `realize` is called on `tuple` execution begins. The function will block until a single complete match is found. The tuple can then be unfolded into the LUT `index` and an `offset`, representing the resolved match.

```
/// Definitions of data stored on host
int64 n_patterns, pattern_size = 4, 8;
Tensor<trit, n_patterns, pattern_size> lut_on_host = read_file();
Stream<bit, pattern_size> data_stream = get_data_stream();

/// Algorithm
Func lut_on_acc = lut_on_host(lut_height, lut_width);
Func (matches, offset) = CompareExact(lut_on_acc, data_stream);
Func tuple = MatchResolver(matches, offset);

/// Schedule
lut_on_acc
    // .reorder(lut_width, lut_height) // Uncomment for column order
  .tile(lut_height, lut_width, i_lut_height, i_lut_width, M_HEIGHT, M_WIDTH)
  .bind(lut_height, CORES)
  .bind(lut_width, MACROS)
  .bind(i_lut_height, M_HEIGHT)
  .bind(i_lut_width, M_WIDTH)

/// Execute (blocks)
Tensor<int64, 2> (index, offset) = tuple.realize()
```

Listing 4.9: Example of network monitoring application that utilizes pattern matching implemented using our programming model. The symbolic variables `lut_height` and `lut_width` are highlighted in red to show that they are bound to the dimensions of the `lut_on_host`.

```
Func<(int64,M),(int64,1)> CompareExact(Tensor<trit, M, N> lut, Stream<bit,N> s0)
{
    return (cimOps.search(lut, s0, EXACT), s0.offset());
}

Func<int64,2> MatchResolver(Tensor<int64, N> matches, Tensor<int64, 1> offset)
{
    return (aluOps.min(matches), offset);
}
```

Listing 4.10: Kernel definitions used in the pattern matching example in Listing 4.9.

```
for bit in data_stream:
  sliding_window = (sliding_window << 1) | bit;
  for lut_entry in lut_height:
    comparisons[...] = CompareExact(lut_on_acc(...), sliding_window);
```

Listing 4.11: Interpretation of calling `CompareExact` in Listing 4.9 without any scheduling applied. Each data_stream bit is shifted into the sliding window, and comparisons against each pattern stored in the accelerator's CIM macros occur sequentially. Indexing is omitted for simplicity.

```
for bit in data_stream:
  sliding_window = (sliding_window << 1) | bit;
  parallel(CORES) for lut_height in CORES:
    parallel(MACROS) for lut_width in MACROS:
        parallel(M_HEIGHT) for i_lut_height in M_HEIGHT:
            comparisons[...] = CompareExact(lut_on_acc(...), sliding_window);
```

Listing 4.12: Interpretation of calling `CompareExact` in Listing 4.9 with the `bind` and `tile` scheduling directives. Each data_stream bit is shifted into the sliding window and pattern comparisons occur in parallel across CIM cores, across CIM macros, and within each CIM macro. Indexing is omitted for simplicity.

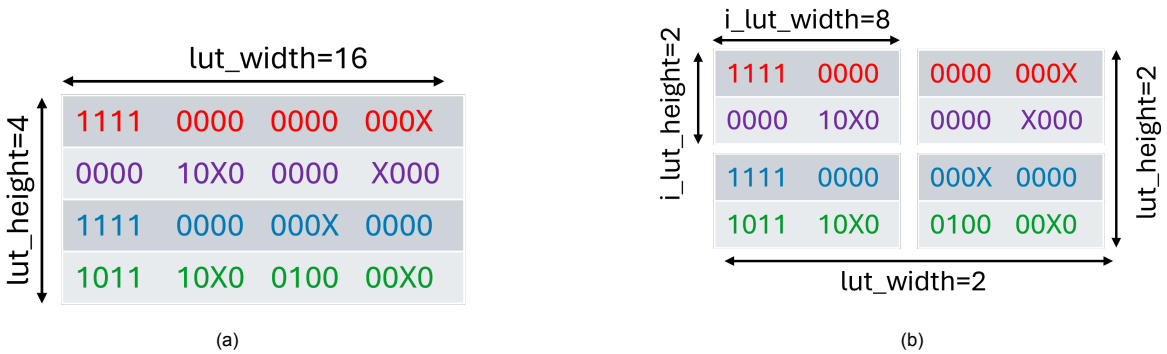


Figure 4.12: Dimensionality of the `lut_on_host` tensor in Listing 4.9 (a) Without applying the `tile` and `bind` scheduling directives (b) With the `tile` and `bind` scheduling directives where the dimension order from innermost to outermost is `M_WIDTH`, `M_HEIGHT`, `MACROS`, `CORES`.

4.2.3. Convolutional Neural Network Inference

We now turn to the implementation of convolutional neural network (CNN) inference. We implemented a minimal binary CNN (BNN) that includes at least one instance of each operation discussed in Section 2.2.3: convolution, max-pooling, sigmoid activation (σ), and a fully connected stage consisting of a single dense layer. The resulting network is shown in Figure 4.13. We assume an architecture similar to ISAAC [52], but one that is programmable and consistent with our computer model.

The first operation we implement is convolution, specifically Conv2D. Unlike the other operations presented so far, convolution requires the use of *img2col*, a data-layout transformation that converts the convolution into a series of MVMs that can be efficiently mapped to CIM-Xbars. To illustrate this, we revisit the transformation from Section 2.2.3, shown again in Figure 4.14, this time highlighting all new symbolic variables used by the kernel that implements *img2col* in Listing 4.13.

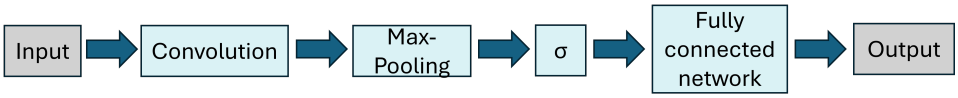


Figure 4.13: Structure of the CNN implemented in this section, containing a single convolution layer, max-pooling layer, sigmoid (σ) activation layer, and a single linear layer in its fully connected network.

Listing 4.13 receives two tensors as input: the original set of filters and the input feature map (IFM), with dimensions shown in Figure 4.14a. The kernel then transforms each tensor and returns the filters and IFM as shown in Figure 4.14b. Each tensor is transformed by applying scheduling directives to it.

The transformation for the `filters` involves fusing the kw, kh, and ic dimensions into a single dimension using the `fuse` scheduling directive. We bind this new dimension to the fk symbolic variable and use it in the return type for the transformed `filters`. The `ifm` transformation is more involved: we first apply `tile` to introduce two new nested dimensions in ih and iw, called $ih\_internal$ and $iw\_internal$, with lengths equal to kh and kw, respectively. Next, we use the `reorder` scheduling directive to change the nesting order of ic. Finally, we apply `fuse` to the remaining dimensions and bind them to the symbolic variables fk and owh. In this case, we can reuse fk, since $ih\_internal$ and $iw\_internal$ have lengths equivalent to kh and kw, making the product $ih\_internal \times iw\_internal \times ic$ equal to $kh \times kw \times ic$.

We note that binding fk and owh was entirely optional and done for readability. By using the wildcard operator in the `fuse` directive, the new dimensions could have been left unnamed, as explained in Appendix A. In the return type, the missing dimensions could then be expressed as products of the symbolic variables involved in the fusion: for fk, this would be $kw \times kh \times ic$, and for owh, it would be $ih \times iw$. Another important distinction is that ih and iw in the return type have different dimensions than their counterparts in the input, since the `tile` directive modifies their sizes.

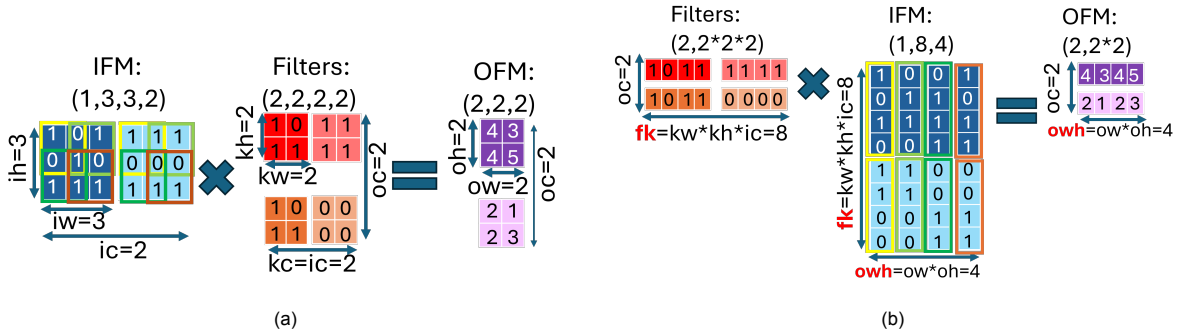


Figure 4.14: Example from Section 2.2.3 used to explain Listing 4.13 (a) Binary `conv2d_nhwc` example with a single-batch IFM and a stride of one. Colored squares on IFM indicate sliding-window positions. (b) `img2col` transformation of Figure 2.11a, resulting in a series of MVMs where filters form the first matrix and the IFM is reshaped into column vectors. We highlight with red the symbolic variables that are bound in Listing 4.13.

```
Func<(bit, oc, fk), (bit, fk, owh)> Img2col(
    Tensor<bit, oc, ic, kh, kw> filters, Tensor<bit, ic, ih, iw> ifm) {
    return (
            filters
                .fuse(ic, kh, kw, fk),
            ifm
                .tile(ih, iw, ih_internal, iw_internal, kh, kw)
                .reorder(iw_internal, ih_internal, ic, iw, ih)
                .fuse(ih_internal, iw_internal, ic, fk)
                .fuse(ih, iw, owh)
    );
}
```

Listing 4.13: Kernel used to implement the *img2col* data-layout transformation, converting from Figure 4.14a to Figure 4.14b. It receives the original IFM and filters, performs the transformation, and returns a tuple of the transformed IFM and filters.

Once *img2col* is performed, the resulting MVMs can be mapped to CIM-Xbars. Figure 4.15a shows an example in which the entire kernel is mapped onto a full CIM-Xbar, and the yellow column vector from Figure 4.14b is applied to produce the first two elements of the output feature map (OFM). In this example, we assume 2×2 memory arrays. As explained in Section 2.2.3, this large MVM can be decomposed into multiple smaller MVMs, after which the partial results are aggregated through addition. Similar to pattern matching, Figure 4.15b and Figure 4.15c illustrate the two possible schedules: a row-wise distribution, shown in Figure 4.16a, and a column-wise distribution, shown in Figure 4.16b.

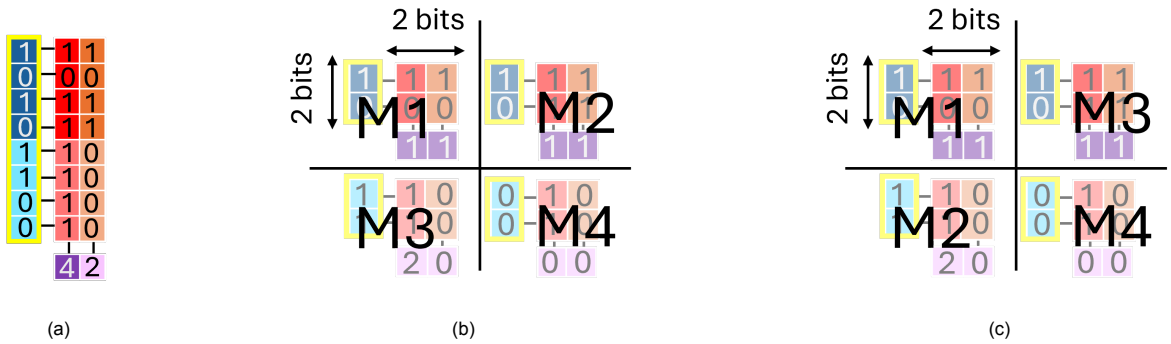


Figure 4.15: (a) Illustration of executing the yellow column vector from Figure 4.14b on a single CIM-Xbar, (b) distributed in a row-wise fashion to 2x2 CIM-Xbars and (c) presents a column-wise distribution.

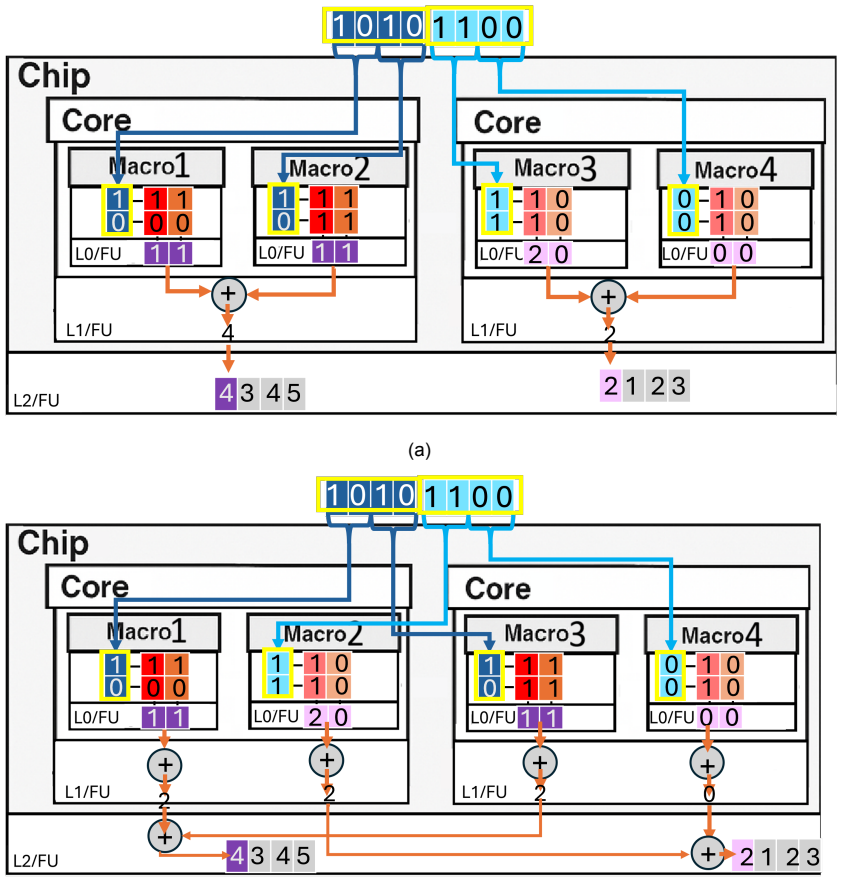


Figure 4.16: Two different schedules for the `Conv2D` operation on the assumed hardware. (a) illustrates the row-wise distribution from Figure 4.15b, in which each output channel is mapped to a single core, and (b) shows the column-wise distribution from Figure 4.15c, where output channels are distributed across cores.

In Figure 4.16a, the row-wise distribution computes each output channel (depicted in a different color) within a single core, reducing the post-processing communication required to aggregate the results. In contrast, the column-wise distribution in Figure 4.16b computes color channels across multiple cores, increasing communication overhead. Therefore, delaying the final addition to the chip level does not appear to offer any immediate benefit.

We explain the remaining operations along with Listing 4.14, which shows the full program for CNN inference. First, we define the required scalars and tensors on the host. For the scalars, we follow the naming convention from Figure 4.14. Next, we apply the `Img2col` kernel introduced earlier in this section, followed by the convolution operation, binding the dimensions of the `filters_on_acc` tensor to fk and s_oc (s means symbolic, to distinguish it from the integer oc) for tiling. Finally, we chain the remaining operations, including max pooling, sigmoid activation, and the fully connected layer.

The kernels used in this example are listed in Listing 4.15. `Conv` performs MVM, where the parameters include a 2D matrix of size $M \times N$ and a stream of vectors of length N. The stride is also N to ensure that a new vector is consumed for each MVM. Subsequently, `MaxPool` reduces the dimensionality of the OFM for each output channel, as illustrated in Figure 4.17a, followed by `Sigmoid`, which applies the sigmoid activation element-wise to its inputs. Finally, `FClayer` performs another MVM, as shown in Figure 4.17c, producing the final output of the neural network.

An important note regarding `MaxPool` is that the entire Tensor must be available before execution can begin. Although the reduction could, in principle, be performed incrementally, doing so would require the programming model to express mutable state. Since Tensors are, by definition, immutable, this is not supported. We leave this to future work.

With regard to the sigmoid function, the output must be binarized before it can be fed to the dense layer. We assume that this binarization is handled as part of the sigmoid unit's functionality and requires no programmer intervention.

Looking at the schedule, it is similar to the pattern-matching approach described in the previous section. We perform tiling to achieve the distribution shown in Figure 4.15b and Figure 4.16a. Since we assume that our accelerator consists of only four CIM macros, the scheduling cannot be fully pipelined when multiple IFM batches are present, as this would require a minimum of five CIM macros: four for the convolution and one for the dense layer.

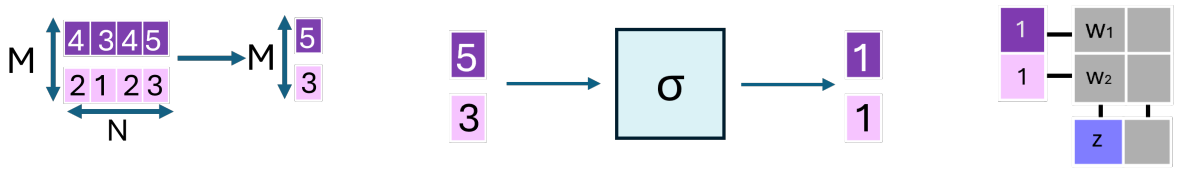
```
/// Definitions of data stored on host
int64 ih, iw, ic, kh, kw, oc, fc_weights = 3, 3, 2, 2, 2, 2, 2;
Tensor<bit, oc, ic, kh, kw> filters_on_host = read_filters();
Tensor<bit, ic, ih, iw> ifm_on_host = read_ifm();
Tensor<bit, fc_weights> fc_layer = read_fc();

/// Algorithm
Func (filters_on_acc, ifm_stream) = Img2col(filters_on_host, ifm_on_host);
Func ofm = Conv(filters_on_acc(s_oc,fk), ifm_stream);
Func max_pool_result = MaxPool(ofm);
Func sigmoid_result = Sigmoid(max_pool_result)
Func fc_result = FClayer(fc_layer, sigmoid_result);

/// Schedule
filters_on_acc
  .tile(fk, s_oc, fk_internal, s_oc_internal, M_HEIGHT, M_WIDTH)
  .bind(fk, CORES)
  .bind(s_oc, MACROS)
  .bind(fk_internal, M_HEIGHT)
  .bind(s_oc_internal, M_WIDTH)

/// Execute (blocks)
Tensor<bit, 1 > result = fcOutput.realize()
```

Listing 4.14: Example of BNN inference implemented using our programming model.

```
Func<int64,M> Conv(Tensor<bit, M, N> tensor, Stream<bit,N, N> stream)
{
    return cimOps.mvm(tensor, stream);
}
```

```
5
6  Func<int64,M> MaxPool(Tensor<bit, M, N> ofm)
7  {
8      return aluOps.max(ofm);
9  }
10
11 Func<int64,M> Sigmoid(Tensor<bit, M> tensor)
12 {
13     return aluOps.sigmoid(tensor);
14 }
15
16 Func<int64,M> FClayer(Tensor<bit, M, N> kernel, Stream<bit,N, N> stream)
17 {
18     return cimOps.mvm(kernel, stream);
19 }
```

Listing 4.15: Kernel used in the BNN application shown in Listing 4.14.



(a) (b) (c)

Figure 4.17: Illustration of the kernels from Listing 4.15. Using the result from Figure 4.14b the (a) max-pooling operation reduces its dimensionality, (b) then applies the sigmoid activation function that produces as many inputs as it receives and (c) shows the final MVM in the dense layer producing the final result z.

5 Discussions

In this chapter, we discuss the challenges encountered throughout the course of this project and outline potential directions for future work. We also reflect on lessons learned during the development of our proposed programming model and how these insights can inform subsequent research. To provide clarity and focus, the chapter is divided into independent sections, each addressing a specific aspect of the work and suggesting possible avenues for improvement or further exploration.

5.1. Comparison with existing work

As discussed in Section 3.1, the ideal CIM compiler or framework—summarized in Table 5.1—should, first, be domain-agnostic, able to express a wide range of CIM applications without being tied to any specific CIM primitive. Second, it should be built on a programming model rather than a purely declarative API, enabling programmers to guide both computation and scheduling. Third, it should expose explicit control over CIM-specific aspects such as resource management.

In Chapter 4, we introduced a CIM programming model and realized it as a DSL. We implemented three CIM applications—integer sorting, network pattern matching, and CNN inference—which, despite coming from different domains, all rely on the CIM-CAM or CIM-Xbar CIM primitives. Thus, while our DSL demonstrates cross-domain expressiveness, the absence of CIM-ALU workloads means it is not yet fully domain-agnostic. Nevertheless, our DSL is based on a programming model that provides explicit scheduling control and exposes CIM-specific features, such as immutable Tensors to minimize analog weight overwrites and tiling to exploit the 2D structure of CIM primitives. Using declarative scheduling primitives, optimizations are also possible—for example, the `bind` directive lets the programmer explicitly map kernels to hardware resources to control parallelism.

In contrast, most existing work focuses on machine learning, deep neural networks (DNNs), or other matrix–vector multiplication (MVM) workloads that rely exclusively on the CIM-Xbar. Notable exceptions include IMDP [51], CINM [44], and C4CAM [72]. IMDP supports both CIM-Xbar and CIM-ALU operations but not CIM-CAM, and because it extends TensorFlow [32]—a framework not designed for CIM—it hides CIM-specific details behind a high-level interface. However, we ultimately lack insight into IMDP's practical expressiveness, as no code examples are available. Both CINM and C4CAM support all three CIM primitives however both of them are rigid API-based compilation flows that lower high-level descriptions down to CIM operations. Hence, programmers have no control over scheduling or optimization and are limited to the operations the compiler designers predicted in advance requiring extending the compiler to add further support.

Our research concludes that CIM programming models should integrate: explicit expression of parallelism, implicit scheduling, hardware generalization, and optimization. While our proposed programming model represents one possible solution for realizing these meta-abstraction, there is no single best solution.Through additional design exploration, alternative abstractions can be developed to achieve the same or different meta-abstractions, each presenting its own trade-offs between performance, portability, productivity and expressiveness.

Name	(1) Application domain	(2) PM or API	(3) Frontend	(4) CIM-specific features
Ideal case	domain-agnostic	PM	–	yes
Ours	no CIM-ALU support	PM	–	yes
OCC[71]	MVM	API (CIM dialect)	Teckyl	no
CINM[44]	limited CAM support	API (CIM dialect)	Pytorch/Linalg/Tosa	no
C4CAM[72]	domain-agnostic	API (CIM dialect)	TorchScript (Pytorch IR)	no
Polyhedral[73]	DNN (MVM)	API	C	compile arguments
TC-CIM[47]	DNN (MVM)	API	Tensor Comprehensions (Pytorch)	no
TDO-CIM[74]	MVM	API	C++	no
CIM-MLC[45]	DNN (MVM)	API	ONNX	architecture parameters
Co-design[46]	ML	API	ONNX	no
CIM-Explorer[56]	BNN/TNN	PM (Halide based)	Larq	strategies per operation
IMDP[51]	no CAM support	PM	TensorFlow	no
PUMA[50]	ML	PM & API	C++/ONNX	no

Table 5.1: Comparison of CIM compilers and frameworks from Table 3.1 enhanced with our contribution highlighted in yellow. The attributes compared are: (1) Any restrictions on application domain, (2) if a programming model (PM) is used or if it is purely API-based, (3) the frontend the user interfaces with (4) whether the user has any control over CIM-specific aspects such as resource management.

5.2. ALU CIM primitive recommended applications

In Section 2.1.2, we mentioned that there are three types of CIM primitives. However, we have only presented applications that utilize CIM-Xbars for MVM or CIM-CAM for parallel search operations. To demonstrate the flexibility of our programming model, originally we planned to implement an application for each primitive. However, we excluded the ALU primitive, as we could not find an accelerator that implements a practical application that uses it.

CIM arithmetic and logic operations occur by first storing the required operands inside the memory array at some specific layout topology. Then computation occurs either inside the memory array itself or during the reading step. In the former case, each operation overwrites memristor cells, reducing their endurance and increasing energy costs. In the latter case, computations are performed during sensing, and—as with other CIM primitives—results are produced at each bit line. Due to the efficiently of the latter approach it is regarded as the ALU primitive.

In our search for accelerators that implement practical applications that use the ALU primitive, we identified the following research. Scouting Logic [79], proposes graph processing and database query applications that have not been explored further. IMDP [51] as mentioned in Section 3.1 is a general-purpose accelerator that provides arithmetic operations and is benchmarked on PARSEC and Rodinia but does not provide code examples. Pinatubo [80] is benchmarked on two practical graph processing and database applications however it uses Computing-Near-Memory (CNM) not CIM as it operates on main memory. Consequently, we could not find any CIM accelerator that employs the ALU primitive in a practical application to replicate.

We briefly considered implementing one of Pinatubo's bitmap-based applications[81, 82] using our programming model, but doing so would require assuming the existence of a CIM accelerator capable of supporting such workloads, along with additional assumptions about the peripheral circuitry and dataflow organization. While this is possible, it would introduce unnecessary complexity and extend beyond the scope of this work. Future work could further explore this idea but it is possible that our proposed model is incompatible with such applications. In that case we would recommend to look into Pinatubo's programming model and see how it can be merged with our current work.

5.3. Defining primitive operations within kernels

One of the challenges we faced was that different accelerators, operating across various application domains, support distinct sets of operations. To design a programming model compatible with all of them, we assume the underlying accelerator supports all possible operations typically associated with CIM such as MVM and parallel search operations. In practice, no such universal accelerator exists, so if our programming model were deployed on real hardware, it would raise an error whenever an unsupported operation was invoked. We do not attempt to compile a comprehensive list of operations, as this would require examining how the same operation is implemented across many different architectures to capture variations in data types, sizes, and communication patterns. Such an analysis is beyond the scope of this work and is left for future research.

Furthermore, as CIM-MLC [45] points out, different accelerators offer different levels of control granularity to the programmer. This raised the question of how fine-grained our proposed operations should be. We decided to adopt high-level instructions that could later be translated into low-level ones by the compiler. What we envision, however, is a model where both high- and low-level instructions can coexist. Specifically, within kernels, programmers should be able to use one of multiple operation sets depending on their needs. These operations could resemble those shown in Figure 5.1, where CIM-MLC's compiler generates instruction for one three hardware layers. However, this might conflict with the dimension-agnostic kernels defined in Section 4.1.4.

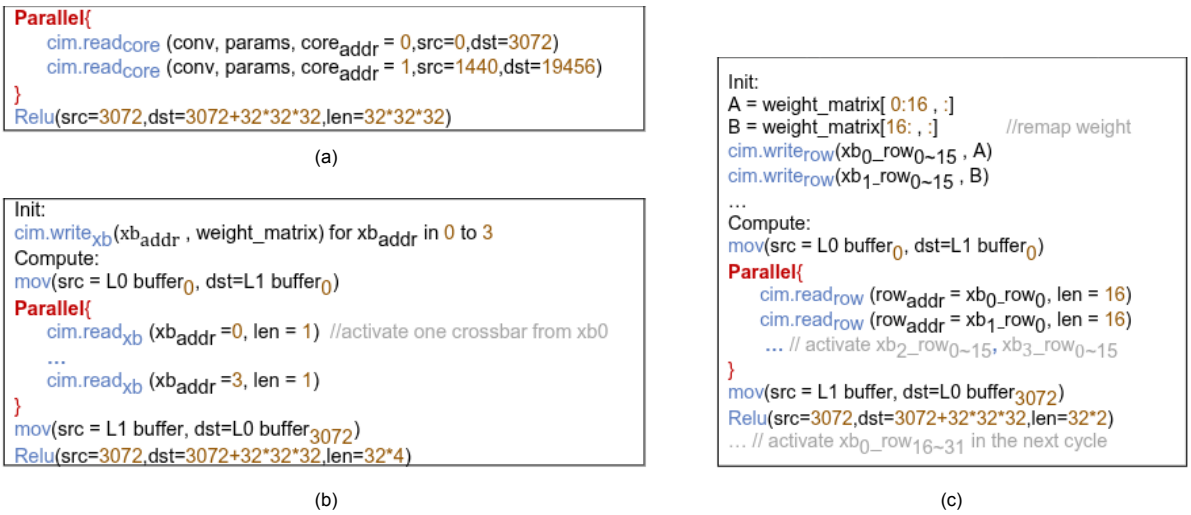


Figure 5.1: Generated code examples for Convolution-Relu of CIM-MLC's compiler [45, Fig. 16]. At each level read and write operations operate at a different granularity where: (a) Core interface operates across cores, (b) Crossbar interface operates across crossbars, (c) Row interface operates within a single crossbar.

5.4. Extending towards heterogeneous CIM platforms

Throughout this manuscript, all CIM accelerators are described as highly specialized ASICs that combine a single type of CIM primitive with a dedicated post-processing circuit. As a result, all implementations discussed are homogeneous in terms of their CIM macros. An interesting direction for future exploration is the introduction of heterogeneity within a single accelerator. This would enable different operations or memory array dimensions to coexist across specialized CIM macros. Using our proposed programming model, and by leveraging the declarative scheduling abstraction, new scheduling directives could be introduced to specify which kernels should be offloaded to each type of CIM macro available in the system.

The immediate question, however, is at which hardware layer of the platform model heterogeneity should be introduced. This could be achieved by incorporating heterogeneous CIM macros, CIM cores, or even CIM chips within a single accelerator. The optimal choice depends heavily on the target applications. While integrating all these variants into a single chip would be prohibitively complex, a more practical approach would be to design multiple coprocessors that are internally homogeneous but heterogeneous across devices. Our programming model could potentially support all of these configurations, laying the groundwork for effectively programming such heterogeneous CIM systems.

5.5. Challenges with mapping tools

Mapping is the process of translating abstract computations, as expressed in our programming model, into concrete hardware operations. Initially, we planned to evaluate our proposed CIM programming model, with mapping by testing how programs written in it can be transformed into efficient and correct executions. This would be achieved by a combination of compilers, simulators or manual translations for one or more different architectures. However, this approach proved to be quite challenging.

Originally, the plan was for this validation to be carried out through another thesis project, in which mapping would have been the core topic and focus. However, that project did not materialize. The

next idea was to use an analytical model to perform the comparisons. An analytical model is a mathematical representation of how a computation behaves, allowing us to estimate metrics such as latency, energy consumption, and total execution time on a given machine. Such a model is being developed in conjunction with this thesis, but it has not yet been completed. In addition, we considered developing a simulator to perform similar comparisons. A simulator is also under development, but it too was not finished at the time of writing. Our next approach was to explore existing simulators to determine whether they could be applied to this work. However, each simulator accepts a different input format, meaning that our implemented applications would need to be converted accordingly to enable comparison. This would effectively require developing a separate compiler for each application, which is not feasible within the time constraints of this project. Therefore, the final plan was to manually lower our programming model into these target instructions.

Table 5.2 lists how each relevant CIM framework or compiler from Section 3.1 conducted its evaluation and which benchmarks were used. We attempted to replicate their experimental setups but encountered several challenges. The most commonly used evaluation platform was the Gem5 simulator [83], typically employed for simulating conventional computer systems but extended in each work to support CIM accelerators. However, OCC [71], CINM [44], and C4CAM [72] provide no means of reproducing their environments beyond brief textual descriptions in their papers. Learning Gem5 in depth and re-creating their customized configurations would have been too time-consuming. TC-CIM [47] and TDO-CIM [74] also rely on Gem5 but only simulate single CIM macro accelerators, which are insufficient for testing our applications that require multiple macros. Co-design [46] and IMDP [51] each employed their own simulators, but the former is confidential while the latter does not specify where its simulator can be obtained. Next, CIM-MLC [45], PUMA [50], and CIM-Explorer [56] use high-level, declarative frontends, making it necessary to interface with their intermediate representations and modify their compilers—an equally time-consuming process. Polyhedral [73], on the other hand, does not use a simulator at all but relies on an analytical model.

Name	Frontend	Benchmarked workloads	Evaluation Medium	Challenges
OCC[71]	Teckyl	μ-kernels, MLP, LSTM, CNN	Gem5 sim	Difficult to replicate
CINM[44]	several DSLs	Same as OCC	Gem5 sim	same
C4CAM[72]	TorchScript	3 CAM-based applications*	Gem5 sim, extended CAMASIM[55]	same
Polyhedral[73]	C	μ-kernels, MLP, CNN	ISAAC analytical model, FPSA tools	no simulator
TC-CIM[47]	TensorCompr	μ-kernels, CNN	Gem5 sim	single crossbar
TDO-CIM[74]	C++	μ-kernels	Gem5 sim	single crossbar
CIM-MLC[45]	ONNX	CNN	Own functional sim, latency, power: other sims	Needed to use IR
Co-design[46]	ONNX	MLP, CNN	Own sim	Confidential
CIM-Explorer[56]	Larq	BNN, TNN	Own sim	Needed to use IR
IMDP[51]	Tensorflow	PARSEC and Rodina benchmark suites	Own sim	No links to sim
PUMA[50]	C++/ ONNX	MLP, LSTM, CNN	Own sim	Needed to use IR

Table 5.2: Comparison of CIM compilers and frameworks. Extends Table 3.1 to include the benchmark workloads and the evaluation medium of each tool. We also include a column that references challenges with using each tool for validation. μ-kernels are small workloads such as simple MVMs, convolutions and variants. *K-Nearest-Neighbor, Hyperdimensional Computing, DNA read mapping.

Ultimately, we abandoned validation through mapping due to the time constraints imposed on this project. While we leave this task for future work, we recommend that future efforts integrate programming model and mapping research in parallel. This approach ensures that abstractions are not only expressive but also efficiently realizable on target hardware, enabling quantitative performance evaluations and meaningful comparisons between different approaches.

5.6. Compiler-assisted concerns

A programming model represents a middle ground between offloading all work to the compiler and assigning full responsibility to the programmer. In practice, this exists on a spectrum, raising the question of which tasks should be handled by the compiler and which should remain under the programmer's control. In this section we discuss which concepts we believe should be offloaded to the compiler for our programming model.

A key feature of our programming model is the use of declarative scheduling directives to influence program execution. One example is the `bind` directive, which parallelizes an iteration bound to a symbolic variable. The programmer can specify how this parallelism should be applied, while the

compiler alone would typically generate a strictly sequential schedule. For clarity, the schedules we demonstrate are interpreted sequentially, as sequential schedules are easier to understand. Some might argue that the compiler could also handle this task. In programming models that separate algorithm from scheduling, such as Legion [38] and Halide [48], the solution is to provide a default schedule based on heuristics. Similarly, a compiler that realizes our programming model could potentially integrate this feature, reducing the need for manually specifying certain directives and decreasing overall code size. However, this comes at the trade-off of increased compiler complexity.

Additionally, the `tile` and `split` scheduling directives form the foundation of our data-parallelism abstraction, as explained in Section 4.1.5. Their purpose is to divide larger computations into smaller, more manageable chunks that can be parallelized. For example, a kernel performing matrix–vector multiplication can be split among multiple workers. This split can occur along two dimensions: horizontally and vertically. As a result, aggregation may be required along one or both dimensions, depending on the operation. In our realized DSL, we assume the compiler is capable of determining the appropriate type of aggregation for each operation. In the case of an MVM, a horizontal split produces partial results that must be combined using addition, while a vertical split only requires gathering the results.

Regarding scheduling directives, we provide a comprehensive list of those used in our programming model in Appendix A. These directives are based on the scheduling primitives provided by Halide [48]. However, Halide offers many more directives, each implemented as a separate function. Some have been adapted to a CIM context, such as the `tile` and `split` directives. Others could be added, which we have not implemented, such as `compute_at` and `store_at`, allowing control over where and when partial results are stored. New directives could also be devised that do not exist in Halide. In this way, scheduling directives balance programmer control with automation, enabling complex optimizations and hardware-specific mappings to be applied transparently while keeping code concise and maintainable. Research of this scope is left for future work.

6

Conclusions

Computing in Memory (CIM) departs from the traditional Von Neumann architecture and its inherent bottleneck. Consequently, programming models rooted in the Von Neumann paradigm, which implicitly encode this bottleneck, are ill-suited for achieving an effective balance of portability, performance, productivity, and expressiveness in CIM. In this work, we analyzed programming models, frameworks, and compilers from both CIM and non-CIM domains and identified four essential meta-abstractions for a successful CIM programming model: explicit expression of parallelism, implicit scheduling, hardware generalization, and optimization. We then realized these meta-abstractions in our proposed programming model, which is based on Halide, an image-processing programming model, and implemented it as a domain-specific language (DSL).

To evaluate the expressiveness of the resulting DSL, we implemented three workloads that benefit from CIM acceleration: integer sorting, pattern matching for network monitoring, and convolutional neural network inference. This demonstrates that our DSL can express applications from diverse domains, though it is currently limited to supporting two of the three CIM memory array structures: CIM crossbars for matrix–vector multiplication (MVM) and CIM content-addressable memory (CAM) for search operations such as table lookups.

In contrast, most prior work avoids using a programming model altogether, favoring an API-based approach that offers little control over scheduling and limited expressiveness. Many of these approaches consider CIM accelerators too complex, often relying on fully automated compiler-based solutions to manage data. Our programming model, however, introduces high-level abstractions that give programmers control over scheduling, allowing developers to define programs in terms of both data-level and task-level parallelism while abstracting unnecessary low-level hardware complexity.

There are two primary avenues for future research. First and most importantly, future work should build on our study by validating the mapping of applications implemented using our DSL on simulators, analytical models, or actual accelerator implementations. This could be achieved by implementing a compiler or by manually lowering the examples to serve as inputs for existing compilers. Once such validation is complete, the results should be compared to relevant prior work, with a quantitative assessment of performance, generated code size, and expressiveness.

Second, research could focus on extending or modifying the proposed model or DSL. For example, an application could be implemented that utilizes the missing CIM memory array structure, the CIM-ALU, which we were unable to validate, demonstrating that the programming model can support all CIM primitives. Additionally, new abstractions could be introduced to implement the recommended meta-abstractions in alternative ways, each offering its own trade-offs.

Overall, the results of this thesis demonstrate that it is possible to design a high-level programming model for CIM capable of expressing applications from diverse domains that benefit from CIM acceleration. While our approach departs from the common practice of fully hiding the complexity of CIM accelerators behind opaque abstractions, we show that functional abstractions can be developed that expose key hardware features to programmers, enabling them to write expressive and hardware-aware code. This work lays the foundation for future exploration of CIM programming models that balance abstraction, control, and performance.

A

Scheduling directives list

Scheduling directive	Description
`split(oldDim, newDim+, int+)`	Breaks down a single dimension into two or more nested dimensions each with a specified integer length.
`fuse(oldDim, oldDim+, newDim)`	Combines two or more nested dimensions into a single new dimension.
`tile(y, x, yi, xi, int, int)`	Breaks two dimensions `x`, `y` into two more where `xi`, `yi` are the two internal dimensions of the sizes specified by the two integers `x`, `y`. Dimensions from innermost to outermost are xi, yi, x and y.
`bind(Dim, x)`	Assign a dimension to a resource to exploit data parallelism. $x \in$ {CHIPS, CORES, MACROS, M_HEIGHT, M_WIDTH}. where M = MACRO
`sequential(Dim)`	The given dimension will be iterated sequentially over time. Useful when there are insufficient resources to perform the operation in parallel and programmers want to influence the outcome.
`malloc_at(x)`	Specify a specific part of the accelerator's memory hierarchy to store intermediate results produce by a functor. $x \in$ {L0, L1, L2, HOST}.
`reorder(Dim+)`	Changes the nesting order of symbolic variables bound to iteration dimensions, letting you control which dimensions are traversed innermost first, specified from left to right
`reduction_order(Dim+)`	Variant of the `reduction` scheduling directive. Can only be applied to a kernel that performs aggregation and returns a functor to a single value. It specifies the order of nested loops where a reduction is performed.

Table A.1: Description of all scheduling directives used throughout this paper for our programming system. The symbol '+' is a quantifier indicating one or more occurrences. Identifiers are symbolic variables bound to dimensions, while integers are denoted with `int`. All x arguments represent an instance of an enumeration type selected for the scheduling primitive. Wildcards marked by `'_'` can be used to omit certain lengths for dimensions that can be calculated by the compiler or symbolic variables that will not be used.

Bibliography

[1] Dayane Reis et al. “In-Memory Computing Accelerators for Emerging Learning Paradigms”. In: *2023 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2023, pp. 606–611. URL: https://ieeexplore-ieee-org.tudelft.idm.oclc.org/document/10044763.

[2] John von Neumann. *First Draft of a Report on the EDVAC*. Tech. rep. University of Pennsylvania, 1945. URL: https://web.archive.org/web/20130314123032/http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf (visited on 06/26/2025).

[3] Calvin Lin and Larry Snyder. *Principles of Parallel Programming*. 1st. USA: Addison-Wesley Publishing Company, 2008. ISBN: 0321487907.

[4] David Patterson et al. “A case for intelligent RAM”. In: *Micro, IEEE* 17 (Apr. 1997), pp. 34–44. DOI: 10.1109/40.592312.

[5] Arthur W. Burks, Herman H. Goldstine, and John von Neumann. *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*. Tech. rep. Princeton, New Jersey: Institute for Advanced Study, June 1946. URL: http://www.cs.unc.edu/~adyilie/comp265/vonNeumann.html.

[6] Wikipedia contributors. *CPU cache, Wikipedia The Free Encyclopedia*. [Online; accessed 28-October-2025]. 2025. URL: https://en.wikipedia.org/w/index.php?title=CPU_cache&oldid=1318472272.

[7] M. Z. Zahedi. “Computation-in-Memory from Application-Specific to Programmable Designs Based on Memristor Devices”. PhD thesis. Delft University of Technology, 2023. URL: https://doi.org/10.4233/uuid:e0a6d2e3-6ec6-4edc-8e6a-5ae931d7dffb.

[8] Kees Vuik. *Future High Performance Computing*. Presentation, DCSE Organization, TU Delft. Presented by Prof. Dr. Ir. Kees Vuik, Slide 12. July 2025.

[9] H.P. Hofstee. “The Next 25 Years of Computer Architecture?” In: Jan. 2009, p. 7. ISBN: 978-3-642-14121-8. DOI: 10.1007/978-3-642-14122-5_2. URL: https://www.researchgate.net/publication/220769144_The_Next_25_Years_of_Computer_Architecture.

[10] Karl Rupp. *42 years of microprocessor trend data*. Accessed: 2025-07-28. Feb. 2018. URL: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[11] D. Englebart. “Microelectronics and the art of similitude”. In: *1960 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. Vol. III. 1960, pp. 76–77. DOI: 10.1109/ISSCC.1960.1157297.

[12] Gordon E. Moore. “Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]” In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 36–37. DOI: 10.1109/N-SSC.2006.4804410.

[13] R.H. Dennard et al. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[14] Temitayo Adefemi. *What Every Computer Scientist Needs To Know About Parallelization*. 2025. arXiv: 2504.03647 [cs.DC]. URL: https://arxiv.org/abs/2504.03647.

[15] Michael J. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[16] R. Duncan. “A survey of parallel computer architectures”. In: *Computer* 23.2 (1990), pp. 5–16. DOI: 10.1109/2.44900.

[17] James C Wyllie. *The complexity of parallel computations*. Tech. rep. Cornell University, 1979. URL: https://ecommons.cornell.edu/bitstream/1813/7502/1/79-387.pdf.

[18] Lawrence Snyder. “Type architectures, shared memory, and the corollary of modest potential”. In: *Annual Review of Computer Science Vol. 1, 1986*. USA: Annual Reviews Inc., 1986, pp. 289–317. ISBN: 0824332016. URL: https://homes.cs.washington.edu/~snyder/TypeArchitectures.pdf.

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

[20] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. “Advances in dataflow programming languages”. In: 36.1 (Mar. 2004), pp. 1–34. ISSN: 0360-0300. DOI: 10.1145/1013208.1013209. URL: https://doi.org/10.1145/1013208.1013209.

[21] William J. Dally, Stephen W. Keckler, and David B. Kirk. “Evolution of the Graphics Processing Unit (GPU)”. In: *IEEE Micro* 41.6 (2021), pp. 42–51. DOI: 10.1109/MM.2021.3113475.

[22] Gajski et al. “A Second Opinion on Data Flow Machines and Languages”. In: *Computer* 15.2 (1982), pp. 58–69. DOI: 10.1109/MC.1982.1653942.

[23] CORPORATE The MPI Forum. “MPI: a message passing interface”. In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing ’93. Portland, Oregon, USA: Association for Computing Machinery, 1993, pp. 878–883. ISBN: 0818643404. DOI: 10.1145/169627.169855. URL: https://doi.org/10.1145/169627.169855.

[24] “IEEE Standard Portable Operating System Interface for Computer Environments”. In: *IEEE Std 1003.1-1988/INT, 1992 Edition* (1988), pp. 1–40. DOI: 10.1109/IEEESTD.1988.8684566.

[25] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[26] Pejman Lotfi-Kamran and Hamid Sarbazi-Azad. “Chapter One - Dark Silicon and the History of Computing”. In: *Adv. Comput.* 110 (2018), pp. 1–33. DOI: 10.1016/BS.ADCOM.2018.03.001. URL: https://doi.org/10.1016/bs.adcom.2018.03.001.

[27] NVIDIA Corporation. *NVIDIA CUDA Programming Guide 1.0*. NVIDIA CUDA Compute Unified Device Architecture. 2007. URL: https://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.

[28] Dmitri B. Strukov et al. “The missing memristor found”. In: *Nature* 453 (2008), pp. 80–83. DOI: 10.1038/nature06932. URL: https://doi.org/10.1038/nature06932.

[29] Nils Voß. *Multiscale Dataflow Programming*. Tutorial document. Maxeler Technologies. 2013. URL: http://home.etf.rs/%5C~vm/os/vlsi/razno/maxcompiler-tutorial%20(3).pdf.

[30] Norman P. Jouppi et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit*. 2017. arXiv: 1704.04760 [cs.AR]. URL: https://arxiv.org/abs/1704.04760.

[31] Richard Hughey. “Programming systolic arrays”. In: Sept. 1992, pp. 604–618. ISBN: 0-8186-2967-3. DOI: 10.1109/ASAP.1992.218541.

[32] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603.04467 [cs.DC]. URL: https://arxiv.org/abs/1603.04467.

[33] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG]. URL: https://arxiv.org/abs/1912.01703.

[34] Wikipedia contributors. *NV1 — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-October-2025]. 2025. URL: https://en.wikipedia.org/w/index.php?title=NV1&oldid=1293548868.

[35] Chuck Pheatt. “Intel® threading building blocks”. In: *J. Comput. Sci. Coll.* 23.4 (Apr. 2008), p. 298. ISSN: 1937-4771.

[36] Pavan Balaji. *Programming Models for Parallel Computing*. The MIT Press, Nov. 2015. ISBN: 9780262332248. DOI: 10.7551/mitpress/9486.001.0001. URL: https://doi.org/10.7551/mitpress/9486.001.0001.

[37] Tetsuya Hoshino et al. “CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application”. In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. 2013, pp. 136–143. DOI: 10.1109/CCGrid.2013.12.

[38] Michael Bauer et al. “Legion: Expressing locality and independence with logical regions”. In: Nov. 2012, pp. 1–11. ISBN: 978-1-4673-0805-2. DOI: 10.1109/SC.2012.71.

[39] Hadi Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 365–376.

[40] Sayeef Salahuddin, Kai Ni, and Suman Datta. “The era of hyper-scaling in electronics”. In: *Nature Electronics* 1.8 (2018), pp. 442–450. ISSN: 2520-1131. DOI: 10.1038/s41928-018-0117-x. URL: https://doi.org/10.1038/s41928-018-0117-x.

[41] Amir Gholami et al. *AI and Memory Wall*. 2024. arXiv: 2403.14123 [cs.LG]. URL: https://arxiv.org/abs/2403.14123.

[42] Dejan Milojicic et al. “Computing In-Memory, Revisited”. In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 2018, pp. 1300–1309. DOI: 10.1109/ICDCS.2018.00130.

[43] John Backus. “Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs”. In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 613–641. ISSN: 0001-0782. DOI: 10.1145/359576.359579. URL: https://doi.org/10.1145/359576.359579.

[44] Asif Ali Khan et al. “CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. ASPLOS ’24. Hilton La Jolla Torrey Pines, La Jolla, CA, USA: Association for Computing Machinery, 2025, pp. 31–46. ISBN: 9798400703911. DOI: 10.1145/3622781.3674189. URL: https://doi.org/10.1145/3622781.3674189.

[45] Songyun Qu et al. “CIM-MLC: A Multi-level Compilation Stack for Computing-In-Memory Accelerators”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, Apr. 2024, pp. 185–200. DOI: 10.1145/3620665.3640359. URL: http://dx.doi.org/10.1145/3620665.3640359.

[46] Joao Ambrosi et al. “Hardware-Software Co-Design for an Analog-Digital Accelerator for Machine Learning”. In: *2018 IEEE International Conference on Rebooting Computing (ICRC)*. 2018, pp. 1–13. DOI: 10.1109/ICRC.2018.8638612.

[47] Andi Drebes et al. “TC-CIM: Empowering Tensor Comprehensions for Computing-In-Memory”. In: *IMPACT 2020 workshop (associated with HIPEAC 2020)*. Informal proceedings. 2020.

[48] Jonathan Ragan-Kelley et al. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *SIGPLAN Not.* 48.6 (June 2013), pp. 519–530. ISSN: 0362-1340. DOI: 10.1145/2499370.2462176. URL: https://doi.org/10.1145/2499370.2462176.

[49] Asif Ali Khan et al. *The Landscape of Compute-near-memory and Compute-in-memory: A Research and Commercial Overview*. 2024. arXiv: 2401.14428 [cs.AR]. URL: https://arxiv.org/abs/2401.14428.

[50] Geraldo F. Oliveira et al. *PUMA: Efficient and Low-Cost Memory Allocation and Alignment Support for Processing-Using-Memory Architectures*. 2024. arXiv: 2403.04539 [cs.AR]. URL: https://arxiv.org/abs/2403.04539.

[51] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. “In-Memory Data Parallel Processor”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 1–14. ISBN: 9781450349116. DOI: 10.1145/3173162.3173171. URL: https://doi.org/10.1145/3173162.3173171.

[52] Ali Shafiee et al. “ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 14–26. DOI: 10.1109/ISCA.2016.12.

[53] Wenyu Sun et al. “A Survey of Computing-in-Memory Processor: From Circuit to Application”. In: *IEEE Open Journal of the Solid-State Circuits Society* PP (Jan. 2023), pp. 1–1. DOI: 10.1109/OJSSCS.2023.3328290.

[54] Rui Liu et al. “MemSort: In-Memory Sorting Architecture”. In: *2024 IEEE 42nd International Conference on Computer Design (ICCD)*. 2024, pp. 28–35. DOI: 10.1109/ICCD63220.2024.00016.

[55] Mengyuan Li et al. *CAMASim: A Comprehensive Simulation Framework for Content-Addressable Memory based Accelerators*. 2024. arXiv: 2403.03442 [cs.AR]. URL: https://arxiv.org/abs/2403.03442.

[56] Rebecca Pelke et al. *Optimizing Binary and Ternary Neural Network Inference on RRAM Crossbars using CIM-Explorer*. 2025. arXiv: 2505.14303 [cs.ET]. URL: https://arxiv.org/abs/2505.14303.

[57] Catherine Graves et al. “In-Memory Computing with Memristor Content Addressable Memories for Pattern Matching”. In: *Advanced Materials* 32 (Aug. 2020), p. 2003437. DOI: 10.1002/adma.202003437.

[58] *What is a Convolutional Neural Network? — nvidia.com*. https://www.nvidia.com/en-us/glossary/convolutional-neural-network/. [Accessed 21-11-2025].

[59] Matthieu Courbariaux and Yoshua Bengio. “BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1”. In: *ArXiv* abs/1602.02830 (2016). URL: https://api.semanticscholar.org/CorpusID:6564560.

[60] Barry de Bruin. *Deep Neural Network Optimization: Binary Neural Networks*. Lecture, Electrical Engineering – Electronic Systems Group. Outline: Introduction, Overview, Designing a Binary Neural Network, Training and Evaluation, State-of-the-Art Models. 2023. URL: https://www.slideserve.com/darleneandre/deep-neural-network-optimization-binary-neural-networks-powerpoint-ppt-presentation.

[61] M.J. Flynn. “Very high-speed computing systems”. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. DOI: 10.1109/PROC.1966.5273.

[62] Eric E. Johnson. “Completing an MIMD multiprocessor taxonomy”. In: *SIGARCH Comput. Archit. News* 16 (1988), pp. 44–47. URL: https://api.semanticscholar.org/CorpusID:40821871.

[63] R. Duncan. “A survey of parallel computer architectures”. In: *Computer* 23.2 (1990), pp. 5–16. DOI: 10.1109/2.44900.

[64] Wikipedia contributors. *Duncan’s taxonomy — Wikipedia, The Free Encyclopedia*. [Online; accessed 12-October-2025]. 2025. URL: https://en.wikipedia.org/w/index.php?title=Duncan%27s_taxonomy&oldid=1304468558.

[65] Rana Umar Nadeem. *From SIMD to MIMD: The evolution of modern computing in gpus and ai*. July 2025. URL: https://medium.com/@ranaumarnadeem/from-simd-to-mimd-the-evolution-of-modern-computing-in-gpus-and-ai-a42fa242dc02.

[66] Tianqi Chen et al. *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*. 2018. arXiv: 1802.04799 [cs.LG]. URL: https://arxiv.org/abs/1802.04799.

[67] Alex Aiken et al. *Legion: A Data-Centric Parallel Programming System*. Accessed: 2025-06-20. URL: https://legion.stanford.edu/.

[68] Matei Zaharia et al. *Spark: Cluster Computing with Working Sets*. Tech. rep. UCB/EECS-2010-53. May 2010. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-53.html.

[69] Panos Rondogiannis and William W. Wadge. “Intensional Programming Languages”. In: 1998. URL: https://api.semanticscholar.org/CorpusID:2023662.

[70] B. Wadge. *We demand data: The story of Lucid and Eduction*. Feb. 2024. URL: https://billwadge.com/2022/07/17/we-demand-data-the-story-of-lucid-and-eduction/.

[71] Adam Siemieniuk et al. “OCC: An Automated End-to-End Machine Learning Optimizing Compiler for Computing-In-Memory”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.6 (2022), pp. 1674–1686. DOI: 10.1109/TCAD.2021.3101464.

[72] Hamid Farzaneh et al. “C4CAM: A Compiler for CAM-based In-memory Accelerators”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS ’24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 164–177. ISBN: 9798400703867. DOI: 10.1145/3620666.3651386. URL: https://doi.org/10.1145/3620666.3651386.

[73] Jianhui Han et al. “Polyhedral-Based Compilation Framework for In-Memory Neural Network Accelerators”. In: *J. Emerg. Technol. Comput. Syst.* 18.1 (Sept. 2021). ISSN: 1550-4832. DOI: 10.1145/3469847. URL: https://doi.org/10.1145/3469847.

[74] Kanishkan Vadivel et al. “TDO-CIM: Transparent Detection and Offloading for Computation In-memory”. In: *2020 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2020, pp. 1602–1605. DOI: 10.23919/DATE48585.2020.9116464.

[75] Facebook and Microsoft. *Onnx/onnx: Open standard for machine learning interoperability*. URL: https://github.com/onnx/onnx.

[76] Chris Lattner et al. *MLIR: A Compiler Infrastructure for the End of Moore’s Law*. 2020. arXiv: 2002.11054 [cs.PL]. URL: https://arxiv.org/abs/2002.11054.

[77] Lorenzo Chelini et al. “Declarative Loop Tactics for Domain-specific Optimization”. In: *ACM Trans. Archit. Code Optim.* 16.4 (Dec. 2019). ISSN: 1544-3566. DOI: 10.1145/3372266. URL: https://doi.org/10.1145/3372266.

[78] Tim Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Sept. 2004. ISBN: 0321228111.

[79] Lei Xie et al. “Scouting Logic: A Novel Memristor-Based Logic Design for Resistive Computing”. In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2017, pp. 176–181. DOI: 10.1109/ISVLSI.2017.39.

[80] Shuangchen Li et al. “Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories”. In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2016, pp. 1–6. DOI: 10.1145/2897937.2898064.

[81] Kesheng Wu. “Wu, K.: Fastbit: an efficient indexing technology for accelerating data-intensive science. Journal of Physics: Conference Series 16, 556”. In: *J. Phys.: Conf. Ser.* 16 (Jan. 2005), pp. 556–. DOI: 10.1088/1742-6596/16/1/077.

[82] Scott Beamer, Krste Asanović, and David Patterson. “Direction-Optimizing Breadth-First Search”. In: *Scientific Programming* 21 (Jan. 2013). DOI: 10.3233/SPR-130370.

[83] Jason Lowe-Power et al. *The gem5 Simulator: Version 20.0+*. 2020. arXiv: 2007.03152 [cs.AR]. URL: https://arxiv.org/abs/2007.03152.