UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Accelerating Memory-Bound Applications with Near-Data Processing: From Architectural Design to Full-System Simulation

João Miguel Morgado Pereira Vieira

Supervisor:	Doctor Pedro Filipe Zeferino Tomás
Co-Supervisors:	Doctor Nuno Filipe Valentim Roma Doctor Gabriel Falcão Paiva Fernandes

Thesis specifically prepared to obtain the PhD Degree in Electrical and Computer Engineering

Draft

June 2024

Funding Institutions
Fundação para a Ciência e Tecnologia (FCT)

Acknowledgments

I wish to thank Fundação para a Ciência e a Tecnologia (FCT) for financially supporting the work of this Ph.D. through the research grant SFRH/BD/144047/2019 (DOI: 10.54499/SFRH/BD/144047/2019) and project 2022.06780.PTDC (DOI: 10.54499/2022.06780.PTDC), as well as Instituto de Engenharia de Sistemas e Computadores - Investigação e Desenvolvimento (INESC-ID) for hosting me and providing me with all the necessary conditions to conduct my research.

I also thank my scientific advisors Prof. Pedro Tomás, Prof. Nuno Roma, and Prof. Gabriel Falcão for all their patience, guidance, and support throughout these last four years. Without them, this Ph.D. would not have been possible. Also, in general, I would like to thank the members of the High-Performance Computing Architectures and Systems (HPCAS) Research Group.

I would also like to leave a special thank you to Ana de Jesus and Vanda Fidalgo, who not only deal with all the usual bureaucracy related to the research work but also go the extra mile and help students with advice.

I thank all my friends and colleagues whom I met during this journey. Each and every one of them left an important mark that I shall never forget.

I thank my family for all the love and support they gave me along the path. To my mom, Maria Rosa, my dad, José Morgado, my sister, Ana Luísa, my grandmother, Maria Eugénia, and my uncle, Afonso: I owe it all to you.

I also want to thank my best friend Sandro, who has always supported me and taught me that anything can be solved with a glass of wine.

Lastly, I thank all those with whom I crossed paths and who have contributed directly or indirectly to my success.

Abstract

The emergence of new data-intensive algorithms in multiple application domains has widened the performance gap between multi-core high-performance Central Processing Units (CPUs) and memory hierarchies. To mitigate this issue, memory subsystems became increasingly more complex, leading to a significant amount of energy being spent across interconnection wires. Furthermore, despite the efforts made, current memory technologies still cannot keep up with the performance provided by CPUs, something that becomes even more impactful when large amounts of unstructured data are involved, resulting in performance degradation and low utilization of the CPUs.

As a result, an opportunity was created for specialized accelerators, which offer significant performance improvements over CPUs for two distinct reasons: (1) they are optimized for a specific set of applications; and (2) they may be installed close to memory devices, becoming Near-Data Accelerators (NDAccs), which grants them a much higher bandwidth and lower latency to memory than the CPU.

The work of this thesis is dedicated to exploring typologies, design techniques and optimizations, and development technologies specifically targeting NDAccs. In a first phase, a Near-Data Processing (NDP) architecture targeting signal-processing applications–the Cache Compute System (CCS)–is designed and prototyped using the gem5 architectural simulator. Results show that the devised system is capable of speeding up common Convolutional Neural Network (CNN) and clustering kernels up to 40.6× when compared with an ARM Cortex-A53 CPU. Furthermore, it also proves to be an efficient alternative to Graphics Processing Units (GPUs).

However, the prototyping process of the CCS raised awareness for a deeper problem: there were virtually no tools supporting the development and evaluation of NDAccs. Therefore, in a second phase of this thesis, a full system simulation solution especially targeting the development and the preliminary evaluation of entire NDP-enabled systems is proposed.

Keywords

Near-Data Processing, Memory-bound, Multi-Level Memory Hierarchies, Hardware Development Framework, Full System Simulation.

Resumo

O aparecimento de novos algoritmos com uso intensivo de dados em múltiplos domínios tornou mais evidente a discrepância de desempenho entre as *Central Processing Units* (CPUs) de alto desempenho e hierarquias de memória. Para mitigar este problema, os subsistemas de memória tornaram-se sucessivamente mais complexos, sendo que atualmente uma quantidade significativa de energia é gasta em transferências de dados. Além disso, apesar dos esforços realizados, as tecnologias de memória atuais ainda não conseguem acompanhar o desempenho das CPUs, o que se torna ainda mais evidente na presença de grandes quantidades de dados não estruturados, resultando na degradação do desempenho e baixa utilização das CPUs.

Consequentemente, os aceleradores especializados tornaram-se um meio de processamento alternativo popular, já que oferecem melhorias significativas de desempenho em relação às CPUs por dois motivos: (1) estão otimizados para um conjunto específico de aplicações; e (2) podem ser instalados próximos da memória, tornando-se *Near-Data Accelerators* (NDAccs), concedendo-lhes uma maior largura de banda e menor latência para a memória do que a CPU.

Esta tese é dedicada a explorar tipologias, técnicas de design e otimizações e tecnologias de desenvolvimento vocacionadas para NDAccs. Numa primeira fase, foi desenvolvida uma arquitetura *Near-Data Processing* (NDP) para aplicações de processamento de sinal–o *Cache Compute System* (CCS). Este dispositivo foi prototipado usando o simulador arquitetural gem5. Os resultados mostram que o sistema projetado é capaz de acelerar *kernels* comuns de *Convolutional Neural Networks* (CNNs) e *clustering* até 40.6× em comparação com um processador ARM Cortex-A53. Além disso, também se revela uma alternativa eficiente em relação às *Graphics Processing Units* (GPUs).

No entanto, o processo de prototipagem do CCS revelou um problema mais urgente: praticamente não existiam ferramentas de suporte ao desenvolvimento e avaliação de NDAccs. Portanto, numa segunda fase desta tese, é proposta uma solução de simulação dedicada ao desenvolvimento e à avaliação preliminar destes sistemas.

Palavras Chave

Processamento Perto dos Dados, Desempenho Limitado pela Memória, Hierarquias de Memória de Múltiplos Níveis, *Framework* de Desenvolvimento de *Hardware*, Simulação de Sistemas Completos.

Contents

List of Figures

List of Tables

Listings

List of Acronyms

ADC	Analog-Digital Converter
AI	Artificial Intelligence
ALU	Arithmetic and Logic Unit
API	Application Programming Interface
BNN	Binary Neural Network
BP	Block Partition
CCS	Cache Compute System
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DAC	Digital-Analog Converter
DB	Database
DDDG	Dynamic Data Dependence Graph
DIMM	Dual Inline Memory Module
DPU	Data Processing Unit
DRAM	Dynamic Random-Access Memory
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
FS	Full System
FU	Functional Unit
GPU	Graphics Processing Unit
HBM	High-Bandwidth Memory
HDL	Hardware Description Language
HLS	High-Level Synthesis
HMC	Hybrid Memory Cube
HRS	High-Resistance State
IPC	Instructions Per Cycle
IR	Intermediate Representation
ISA	Intruction Set Architecture
kNN	k-Nearest Neighbors
LCU	Livia Cache Unit
LLC	Last-Level Cache
LMU	Livia Memory Unit
LRS	Low-Resistance State
LSU	Load/Store Unit
LUT	Lookup Table
MAC	Multiply-Accumulate
ML	Machine Learning
MMU	Memory Management Unit
NDAcc	Near-Data Accelerator
NDCU	Near-Data Compare Unit
NDP	Near-Data Processing
NFA	Nondeterministic Finite Automaton
NN	Neural Network

NoC Network-on-Chip

NVM Non-Volatile Memory

OoO Out-of-Order

OS Operating System

PCM Phase Change Memory

PCU PEI Computation Unit

PEI PIM-Enabled-Instruction

PE Processing Element

PIM Processing in Memory

PI Programming Interface

PMU PEI Management Unit

PNM Processing Near Memory

RAM Random Access Memory

ReLU Rectified Linear Unit

ROB Re-Order Buffer

RRAM Resistive Random-Access Memory

RTL Register Transfer Level

SE System Emulation

SIMD Single Instruction Multiple Data

SoC System on Chip

SRAM Static Random-Access Memory

STT-RAM Spin-Transfer Torque RAM

TLB Translation Lookaside Buffer

TSV Through-Silicon Via

UVE Unlimited Vector Extension

1

Introduction

Contents

With the emergence of data-intensive algorithms across several application domains, such as text, voice, image, and video processing, clustering and classification, and neural networks, the inability of memory subsystems to feed data at the required rate and take advantage of the full parallel computing capabilities of high-performance Central Processing Units (CPUs) became a main bottleneck [1]. Moreover, on big data applications characterized by unstructured datasets and irregular memory accesses, where the re-utilization of data in cache is low, data transfers cause significant energy consumption overheads, which can be as high as $64\,\%$ [2]. This contrasts with the $1\,\%$ fraction of energy used for the actual computation [3]. To overcome these limitations, accelerators and co-processors operating close to where the data is actually stored (i.e., memory devices) have been proposed and adopted over the years [4]. Such solutions, henceforth designated Near-Data Accelerators (NDAccs), benefit from a much higher bandwidth to the memory than the CPU, allowing the acceleration of memory-bound workloads.

One of such strategies, designated Processing in Memory (PIM) [5], was first introduced during the '80s, and aims at re-purposing existing Dynamic Random-Access Memory (DRAM) resources to enable bit-line computation. Later, this paradigm was extended to Static Random-Access Memory (SRAM), allowing to perform active computation on caches [3, 6]. While devices using this technique usually offer massive parallelism, being able to simultaneously process whole memory lines in a single clock cycle, they are only capable of performing very simple bit-wise operations and usually require thousands of computation cycles (and complex control circuitry) to implement arithmetic operations using bit-line computation. Moreover, due to its analog nature, bit-line computation is error-prone and is only viable if a small number of memory lines are combined during an operation. As a result, the adoption of pure PIM-based solutions usually requires the programmer to be aware of the underlying memory structure, which makes it unfeasible to achieve a universal programming paradigm for PIM. Furthermore, for most applications, the actual benefits provided by PIM are often limited, being only suitable for a rather small set of programs.

Another similar processing paradigm that recently became popular is Resistive Random-Access Memory (RRAM)-PIM. Solutions using this computation technique rely on the resistive properties of RRAM to implement analog or digital computation. By controlling the impedance of the RRAM cells and their input voltage, it is possible to implement analog logic operations as well as analog multiplication–the main operation required by Neural Networks (NNs) [7]. However, the RRAM fabrication process is known to have significant process variations [8], which makes memristors uneven across devices and also within the same device. Moreover, the impedance of the memristors is affected by the occurrence of temperature fluctuations resulting from the nor-

mal operation of the circuit, which also directly impacts the computation results [9]. Therefore, a significant error is introduced in RRAM-based analog computations, ultimately invalidating the results. Nevertheless, by combining several memristors, it is possible to implement digital bit-wise operations [10, 11], which are not as susceptible to the errors that occur in the analog domain. However, these digital computations are still limited to the same simple logic instructions achieved with PIM, worsened by the fact that since multiple memristors are required to implement a single bit-wise operation, less parallelism can be achieved in comparison with analog DRAM-based PIM.

To circumvent these limitations, a different approach has been adopted by Processing Near Memory (PNM) solutions, featuring fully digital architectures directly connected to the memory devices [12–14]. Although these usually benefit from a lower bandwidth than those implemented within the memory chips, their architectures are not as restricted by the memory technology, resulting in more versatile devices, capable of efficiently processing complex workloads, while still benefiting from a higher bandwidth to the memory than the CPU. Moreover, these co-located devices also provide the means to reduce the energy consumption across wires, as they are physically much closer to the memory than conventional CPUs. Naturally, these devices have a broader scope than PIM-based solutions, and entire general-purpose co-processors can be implemented using this technology.

However, there are still very few general-purpose NDAccs, with most proposals relying on specific programming paradigms and requiring the programmer to have knowledge of the memory architecture at a deep level. Furthermore, such architectures frequently depend on the specialization of the underlying memory subsystem–making it specialized for Near-Data Processing (NDP)–, often rendering it underperformant for classic computation using the CPU and the memory hierarchy as intended. **Therefore, it is crucial to develop general-purpose NDAcc architectures that mitigate these issues, supporting standard programming paradigms (less dependant on the architecture of memory devices and data placement), and requiring less specialization of the memory hardware (providing significant performance benefits while not jeopardizing the classic utilization of the CPU and the memory hierarchy).**

Another important obstacle to the development of NDAccs concerns their evaluation. This aspect is critically different from standard accelerators since the behavior of the memory devices and the cooperation with the host CPU highly affects their performance. Therefore, assessing the architecture of an NDAcc alone tends to poorly reproduce its real performance when integrated with a standard processing system. In fact, the most viable and accurate approach to validate and predict the performance benefits of NDAccs often requires the evaluation of the entire system.

However, such a task is not simple, and there are no standard mechanisms to accomplish it.

A valid approach is to directly prototype NDAccs at Register Transfer Level (RTL) using Hardware Description Languages (HDLs), which is generally complex, expensive, and time-consuming, particularly when the intended goal is to simply evaluate the potential of an idea or to adjust the parameters of an architecture still under consideration. Furthermore, unless the NDAcc is actually fabricated alongside a memory device (e.g., as a chiplet integrated into an High-Bandwidth Memory (HBM) chip), such methodologies still do not consider the influence of the normal operation of the memory subsystem in the performance of the NDAcc, which tends to lead to over-optimistic results and conclusions.

Alternatively, some works re-purpose existing tools to estimate the performance of custom systems. For example, in [15], the authors use a Graphics Processing Unit (GPU) simulator to estimate the performance of a PIM device. Other works extend existing simulators to partially support specific NDAccs [16–20]. However, these artifacts are often designed having a specific NDAcc in mind and are hardly suitable to evaluate different devices. Furthermore, they often require post-simulation analysis [16, 19], which is not compatible with the simulation of a real-time scenario where one or more NDAccs operate in parallel with the CPU. In addition, they frequently disregard the necessary mechanisms to integrate NDAccs with the remaining system, such as the physical infrastructures required to connect these devices with the CPU and the memory subsystem. As a result, they frequently leave important questions unanswered, such as how to cope with the CPU's virtual memory system, which is not only crucial for the system to operate but is also an important factor that has to be taken into account when assessing the performance and energy requirements of the overall processing system and, in particular, the NDAccs.

Hence, it is fundamental to develop standard mechanisms specifically tailored to easily evaluate the benefits of new NDAccs without undergoing the expensive development steps required by traditional hardware development methods, while still allowing to obtain accurate results.

1.1 Objectives

Driven by the growing importance of NDP across a wide range of scientific fields, as well as the need for a standard procedure to develop and validate new NDAccs (particularly targeting early development stages), the main goal of this thesis is to identify the nuclear challenges in this process and propose methods and tools to tackle them. In particular, this thesis aims at (1) developing general-purpose-oriented NDAcc architectures, evaluating their performance benefits over

standard processing systems such as CPUs and GPUs, and determining how such architectures can be further optimized to increase performance; and (2) developing a set of tools specifically targeted at the development and early-stage evaluation of NDAccs.

Hence, the following objectives were envisaged:

1. **Vectorial NDAcc architecture for signal processing applications:** The development of a general-purpose PNM architecture particularly optimized for signal processing applications, such as Convolutional Neural Networks (CNNs), not only constitutes an optimization exercise to fully exploit the parallelization potential of such applications and the benefits allowed by NDP, but also an opportunity to explore and understand the main obstacles in the early phases of development and validation of this class of devices. It is particularly important to establish specific methodologies to integrate the NDAcc with both the CPU and the memory hierarchy, mechanisms to cope with the virtual memory management system of the CPU, and NDP programming paradigms.

2. **Full-stack toolchain targeting the development and preliminary assessment of NDAccs:** Based on the findings of the previous objective, it is also the goal of this thesis to propose a set of tools and procedures to facilitate the development and preliminary evaluation of NDAccs, without resourcing to formal hardware development technologies, which are not only complex and time-consuming but also expensive (hence, unsuitable for early-stage development, or simply testing an idea). Specifically, the devised toolchain aims at offering an integrated solution to model NDAccs, integrate them with a (highly configurable) conventional processing system, and evaluate them using conventional software benchmarks.

Figure 1.1 illustrates the system stack of an NDP-enabled processing system, identifying its different components and highlighting the modules targeted by the work of this Ph.D. thesis. While objective **1.** focuses on the development of the actual NDAcc architecture (NDP architecture block), objective **2.** involves the creation of all mechanisms supporting the integration of NDAcc architectures with conventional processing systems (identified in red). Namely, it envisages the definition of an architectural framework acting as a boilerplate for building custom NDAccs, simulation scripts enabling the simulation of bare-metal or standard processing systems featuring an Operating System (OS), templates for developing the required NDAcc device drivers–to enable communication between an application being executed in an OS environment and the NDAcc–, and libraries and a programming paradigm to use NDAccs from the host applications.

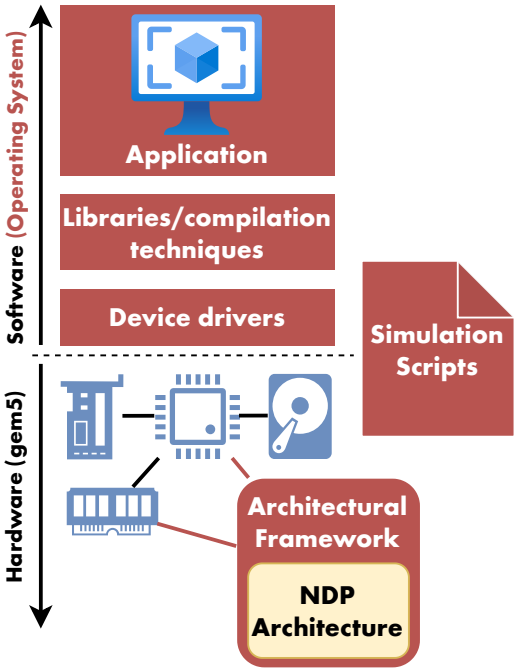


Figure 1.1: Full stack of a NDP-enabled system highlighting the components targeted by the research performed in this Ph.D. thesis.

1.2 Thesis Contributions

The work of this Ph.D. resulted in the publication of five scientific articles in prestigious international journals and conferences. Namely, in the context of objective **1.**, the following publications were achieved:

- **João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. Processing Convolutional Neural Networks on Cache. In International Conference on Acoustics, Speech, and Signal Processing (ICASSP), pages 1658–1662. IEEE, 2020**

  This paper presents an NDAcc architecture tailored for the execution of CNNs. The devised architecture is an extension of the Cache Compute System (CCS) presented in [12], and it includes specific hardware and commands to accelerate operations commonly used CNNs, such as convolution and pooling. The novel NDAcc was modeled and evaluated using the gem5 architectural simulator [21] and the obtained performance metrics were compared with those obtained while executing the same benchmarks in CPUs and GPUs.

- **João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. A Compute Cache System for Signal Processing Applications. Journal of Signal Processing Systems (JSPS),**

93(10):1173–1186, 2021

This work is a natural extension of the aforementioned publication [13]. A new subset of commands was added to the CCS in order to support generic signal-processing applications, broadening the scope of the devised architecture. In addition, a fully detailed specification of the presented architecture is provided (which was not possible in [13] due to the conference's four-page constraint), as well as the full Intruction Set Architecture (ISA) and description of each supported command.

Regarding objective **2.**, the following articles were published:

- **João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. gem5-ndp: Near-Data Processing Architecture Simulation From Low Level Caches to DRAM. In International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD), pages 197–200. IEEE, 2022**

  This paper describes a simulation framework specifically aimed at the development and early assessment of NDAccs. The devised framework, based on the widely-established gem5 architectural simulator [21], consists of a layer on top of gem5 that deals with the simulator's low-level communication protocols and lets the developer focus on the development of the actual NDAcc architecture. In particular, it includes an architectural model that acts as a boilerplate for new NDAcc architectures, generating (highly configurable) Load/Store Unit (LSU), Memory Management Unit (MMU), and Programming Interface (PI) hardware blocks out-of-the-box, as well as bare-metal simulation scripts that allow simulating the developed NDAccs when integrated with a standard processing system.

- **João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. gem5-accel: A pre-rtl simulation toolchain for accelerator architecture validation. IEEE Computer Architecture Letters (CAL), 23(1):1–4, 2024**

  Based on the aforementioned framework, the toolchain proposed in this paper extends the previous work by allowing the simulation of custom NDAcc architectures (and accelerators in general) integrated with complex processing systems executing an OS. In addition, the LSU implemented by the architectural model was redesigned to further improve performance and overcome limitations inherent to the simulator. To the author's knowledge, the proposed tool is the first of its kind, allowing the accurate evaluation of hardware accelerators at the cost of a fraction of the time had an RTL-based evaluation been adopted.

- **João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. Ndpmulator: Enabling full-system simulation for near-data accelerators from caches to DRAM. IEEE Access, 12:10349–10365, 2024**

  This paper constitutes a natural extension of the previous publication. It explains in detail the newly adopted approach for the architectural framework (specifically the LSU) as well as the corresponding simulation scripts. In particular, the Full System (FS) mode is explored in detail. Furthermore, three existing NDAccs were modeled and evaluated using the devised toolchain and the obtained results were compared with those presented in the original papers. Results show that the proposed toolchain allows for the accurate modeling of the considered architectures in a fraction of the time (when compared with classic development techniques), while still allowing to obtain realistic results.

1.3 Organization of this Document

The remainder of this document is organized as follows: Chapter 2 summarizes the most relevant proposals and the state of the art on NDP. In chapter 3, a novel PNM architecture that leverages the data locality and high bandwidth enabled by the Last-Level Cache (LLC) of an ARM processing system to accelerate signal-processing applications is proposed. Chapter 4 presents a new toolchain specifically targeted at the development, modeling, simulation, and early evaluation of custom NDAcc architectures. Finally, chapter 5 summarizes the core findings of this Ph.D. thesis and lays possible future research paths, concluding this document.

2

Background and State of the Art

Contents

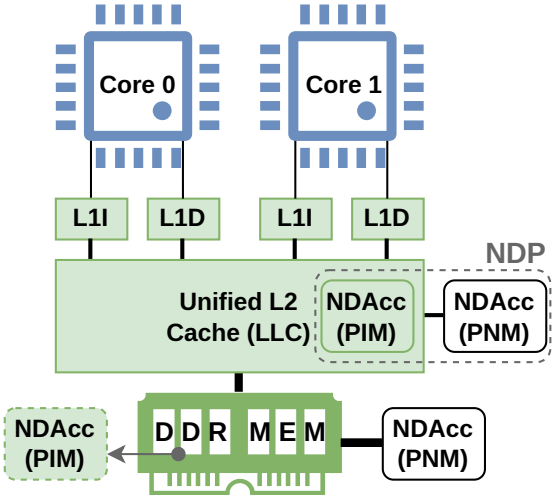


Figure 2.1: Difference in the location of PIM and PNM NDAccs in a processing system.

Driven by the need to fill the gap between the high-performance Central Processing Units (CPUs) and the complex and comparatively underperforming memory hierarchies, Near-Data Processing (NDP) aims at reducing the distance between storage and processing devices, by placing computational resources, designated by Near-Data Accelerators (NDAccs), close to where the data is actually stored (i.e., memory devices) [25, 26].

Historically, the first NDP proposals consisted of repurposing existing resources within the memory chips to perform active computation. Hence, such solutions co-existed in the same space as the memory cells (i.e., inside the memory chips), between the arrays of memory cells and the memory controller. Therefore, the NDP paradigm used by these devices was named **Processing in Memory (PIM)** [27].

However, PIM is highly constrained by the preexisting internal architecture of memory devices, leaving little design space for NDAccs. Therefore, another NDP paradigm was created where NDAccs are installed outside the memory controller (or even in a different chip), further away from the memory cells, where restrictions are lighter, providing far greater design space. This paradigm was designated by **Processing Near Memory (PNM).** Figure 2.1 illustrates a processing system and its corresponding memory hierarchy that has both PIM and PNM NDAccs to exemplify where these accelerators may be located.

Naturally, both approaches present benefits and drawbacks. While PIM is characterized by its massive bandwidth while accessing data, as well as more parallel processing opportunities, it is highly limited by the preexisting memory architectures, narrowing its scope and discouraging its adoption. In contrast, PNM allows much higher design freedom, with the possibility of implementing entire co-processors close to the memory devices, but at the cost of lower bandwidths than

those enabled by PIM.

Hence, to better understand the potential benefits and constraints of PIM and PNM, the following subsections explore these two NDP paradigms and detail the most commonly used processing techniques associated with them.

2.1 Processing in Memory

Being the first NDP paradigm ever proposed (and the one that leverages the lowest-level properties of memory devices), PIM highly depends on the technology in which the memory devices are based. In other words, different types of memory cells (capacitors, transistors, memristors) have distinct physical properties, which determine how they can be used to implement active computation [28–36]. For example, the combined charge of multiple capacitors of a Dynamic Random-Access Memory (DRAM) can be interpreted in the analog domain as a logical operation [37]. Differently, by controlling the impedance and input voltage of a memristor, it is possible to implement an analog multiplication operation [38]. Therefore, to understand how PIM actually works, it is necessary to distinguish between each memory technology, as these result in different ways to perform PIM.

2.1.1 Dynamic Random-Access Memory

DRAM is a type of volatile memory used to temporarily store data while it is being used. Unlike non-volatile memory technologies, such as Resistive Random-Access Memory (RRAM), DRAM needs to be periodically refreshed to maintain the stored data, as it loses information rapidly when the power is turned off. It stores each bit of data in a separate capacitor within an integrated circuit. The main advantages of DRAM are its structural simplicity (only one transistor and one capacitor are required per bit) and cost-effectiveness, making it the prevalent choice for the main memory across a wide range of processing systems.

One Dual Inline Memory Module (DIMM) of DRAM [39–41] contains several dies, which encapsulate one or more memory arrays, as shown in Figure 2.2. A memory array is a grid composed of storage cells, which are capacitors charged to produce a logic ’1’ or a ’0’. These capacitors are only able to keep their electrical charge for a short period because of the current leakage, therefore requiring periodic recharging. In a memory array, the storage cells of the same column are connected to the same bit-line through a transistor, and all the transistors in the same line are attached to a common word-line. When the voltage on a word-line goes high, all the transistors in that row act as closed switches, connecting the storage cells to the bit-lines. Then, the capacitors

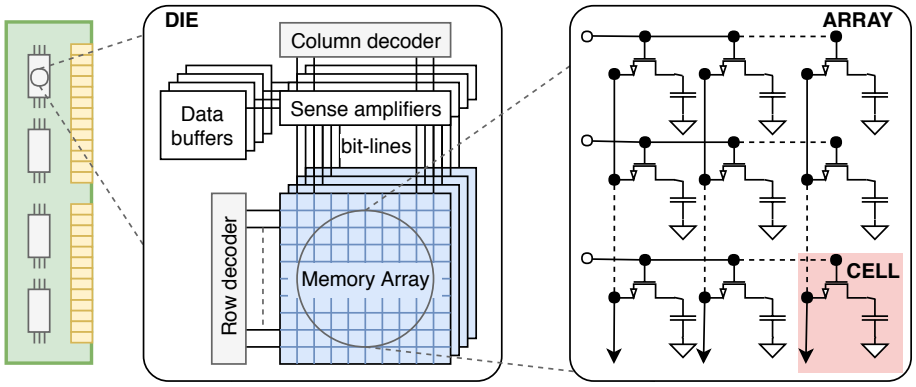


Figure 2.2: DRAM hierarchy. Each DIMM of DRAM usually has multiple dies, which are composed by one or more memory arrays. A memory array is a grid of storage cells that can be read by activating the correspondent word line, which closes the circuit formed by the cell (capacitor) and its associated bit line.

share their charge with the bit-lines, which increases or decreases their voltages above/below $V_{DD}/2 + \delta$. Sense amplifiers connected to each bit-line then detect and reinforce (towards V_{DD} or $0\,\mathrm{V}$) the voltage deviations. Depending on the detected deviation, the sense amplifiers pull the voltage of the bit-line to a full swing of V_{DD} (the maximum voltage) or to $0\,\mathrm{V}$, to represent the logic '1' and '0', respectively. This also allows recharging the capacitors while they are still connected to the bit-line through the respective transistors, as the sharing of their charges with the bit-lines destroys the data that was previously stored (read-destructive). In summary, the process of reading one line of a DRAM memory array is composed of four phases, as illustrated in Figure 2.3:

1. Pre-charge: Before activating the word-line which connects all the cells of a given row of the memory array to the bit-lines, the bit-lines are pre-charged to a reference voltage, e.g., $V_{ref} = V_{DD}/2$.

2. Access: When a voltage is applied to a word-line, the access transistors in that row close the circuit formed by the storage cells (capacitors) and the bit-lines, and the charge of each cell is shared with the respective bit-line. If the charge in the storage cell represents '1', the sharing process increases the voltage on the bit-line from V_{ref} to V_{ref}^{+}, whereas if it represents '0' the voltage on the bit-line will deviate to V_{ref}^{-}.

3. Sense: After the word-line is activated and the memory cells share their charge, the sense amplifier detects the deviations induced to the bit-lines. The time required to perform the sensing is called *sensing-time* or *sensing-delay*, and represents a significant portion of the total memory access time.

4. Restore: After the bit-lines are driven to the respective maximum or minimum voltage values,

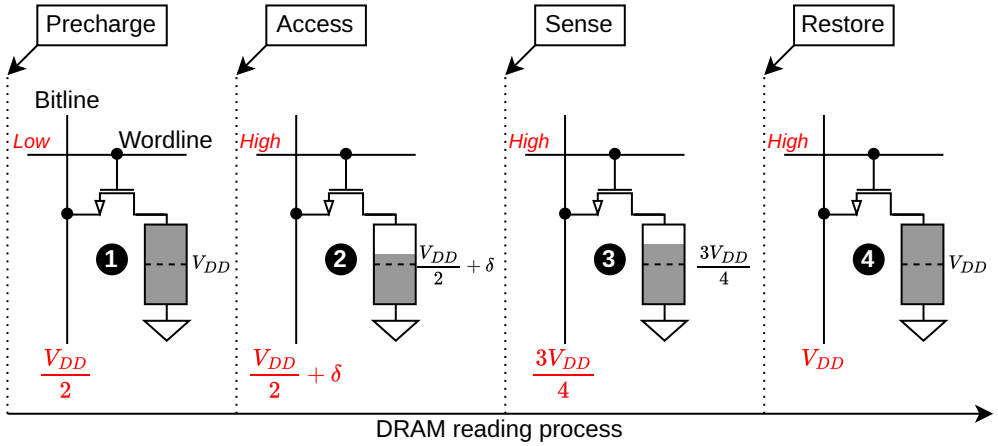


Figure 2.3: The reading process of a DRAM line can be divided into four phases: (1) Pre-charge: the bit-lines are driven to $V_{DD}/2$; (2) Access: the DRAM cells are connected to the bit-lines and share their charge; (3) Sense: the sense amplifiers reinforce the detected deviations and interpret them as logic '1's or '0's; and (4) Restore: the original charge of the DRAM cells is restored.

the selected word-line remains active, and the fully driven bit-lines' voltages restore the charges in the capacitors through the access transistors. Meanwhile, the voltage value on the bit-line can be driven out of the sense amplifier circuit to provide the requested data. In this manner, the contents of a row can be accessed concurrently with the row restoration process.

In the conventional operation of a DRAM, only one word-line is activated during the *access* stage. However, if several rows are activated simultaneously, the voltage of a given bit-line is influenced by the several charges of the storage cells associated with the activated rows, and the sensed result will be the combination of the loads. Let us consider the case were two word-lines are activated at once and the charges of two storage cells associated with a given bit-line are combined. In the instant before the word-lines are activated, all the bit-lines are pre-charged with a given voltage V_i. Right after the word-lines are activated, the loads of two cells are shared with their associated bit-line, and three cases are possible:

1. *Both cells represent a logic '1':* The sensed voltage is given by $V_s > V_i + 2\delta$, where δ is the minimum deviation such that the sense amplifier can detect a logic '1';
2. *Both cells represent a logic '0':* The sensed voltage is given by $V_s < V_i - 2\delta'$, where δ' is the minimum deviation such that the sense amplifier can detect a logic '0';
3. *One cell represents a logic '1' and the other a logic '0':* The sensed voltage is given by $V_s \approx V_i$.

Considering a threshold of $V_{th}^{+} = V_i + 2\delta$, the sensed value can be translated into the result

of the logic *and* operation between the two bits stored in the cells connected to the bit-line (if the detected value is higher than V_{th} the result of the operation is '1'). On the other hand, if the reference is $V'_{th} = V_i - 2\delta'$, the sensed value represents the result of the logic *or* operation between the two bit operands (if the sensed value is higher than V'_{th} the result of the operation is '1').

It is also possible to activate more than two lines simultaneously, which allows to perform the logic *and* and the logic *or* operations with even more operands. However, increasing the number of simultaneously activated word-lines also increases the probability of data corruption. Moreover, due to the read-destructive nature of the DRAM, the operands are destroyed after concluding the bitwise operation, i.e., the cells that contained the operands will end up with the result of the operation. Therefore, if the operands are to be maintained, they have to be previously copied.

Some recent works use bit-line computation to perform logic operations other than the elementary *and* and *or*, such as *xor* and *inv* [42], or even carry-less multiplication or search [3]. Others, also integrate dedicated logic to perform even more complex operations, not doable using only bit-line computation, such as Euclidean distance and dot product [43].

As an example, Seshadri et al. [44] proposed a fast bulkwise *and* and *or* using bit-line computation. The principle of this mechanism lies in activating three word-lines simultaneously, making the deviation on each bit-line after the *sharing* phase to be towards the majority value of the three cells. Two out of the three activated rows are the operands, while the third row is for control (to determine whether the performed operation is an *and* or an *or*). Before performing the computation, the control row is homogeneously initialized with '0' or '1'. After the simultaneous activation of the three lines, eight different outcomes are possible, as shown in Table 2.1. If the control row is initialized with '0', the outcome of the *sense* stage is the result of the logic *and* operation of the operands, whereas if it is initialized with '1', the outcome is the result of the logic *or* operation. In general, the result of the logic operation performed in memory is given by $R(A + B) + \overline{R}(AB)$, where R represents the control row and A and B are the operands [37, 45]. The replacement of the operands with the result of the bitwise operation is solved by the authors by previously copying the source data to temporary rows using a customized and efficient mechanism of their creation, the RowClone [46]. As the computation is performed right after copying the operands to the temporary positions, the storage cells are fully refreshed in the *access* phase of the process, which is crucial, since the deviation on the bit-line caused by the simultaneous activation of the three lines is lower than when only one cell is connected. To support the devised system, the CPU is required to implement specific instructions. In [44], the mechanism was tested and compared with

a system featuring an Intel Core-i7-4790K and 16 GB DDR3-1333, showing speedups as high as $10\times$ and energy consumption reductions up to $50.5\times$.

More recently, Ferreira et al. [47] have proposed a different approach, pLUTo, where traditional bit-line computation is partly replaced by Lookup Table (LUT) queries. The key idea is to leverage the bit-line and word-line structure of DRAM arrays to perform parallel computations directly within the memory. Instead of storing data in a conventional way, pLUTo employs LUTs that associate input values with desired output values. In general, the operation principle of pLUTo has two main phases: (1) for a given complex function whose output only depends on its input, pLUTo populates a LUT with all the possible results (which have to be pre-computed); then (2) the corresponding operations can be replaced by memory readouts of that LUT, which are faster and more energy-efficient. Notably, pLUTo does not require modifications to the DRAM arrays themselves, making it potentially easier to adopt in existing DRAM technologies. In addition, pLUTo makes use of improvements already proposed by previous work to classic DRAM devices, such as RowClone-FPM [46], LISA-RBM [48], MASA [49], Ambit [37], and DRISA [50], enabling efficient copy and shift mechanisms as well as massive bit-line computing capabilities. The work highlights the potential of exploiting the parallelism inherent in DRAM arrays for enabling efficient computation within memory, thereby reducing data movement and improving performance and energy efficiency.

The authors demonstrate the applicability of pLUTo by implementing various computational primitives, such as vector-vector operations, matrix-vector multiplication, and neural network computations, showing that pLUTo can achieve significant performance improvements and energy efficiency compared to conventional CPU-based or Graphics Processing Unit (GPU)-based implementations for certain workloads. The presented results show that pLUTo offers a speedup of

Table 2.1: Possible outcomes of the simultaneous activation of three DRAM word-lines. R represents the reference row that is homogeneously initialized with the logic '0' or '1' to control which bitwise operation is performed (*and* or *or*), and A and B designate the operands. To simplify the representation, the entries of the table contain the logical values represented by the actual charges contained in the storage cells and voltage sensed by the amplifier.

R	A	B	Sensed value	
'0'	'0'	'0'	'0'	and
'0'	'0'	'1'	'0'	
'0'	'1'	'0'	'0'	
'0'	'1'	'1'	'1'	
'1'	'0'	'0'	'0'	or
'1'	'0'	'1'	'1'	
'1'	'1'	'0'	'1'	
'1'	'1'	'1'	'1'	

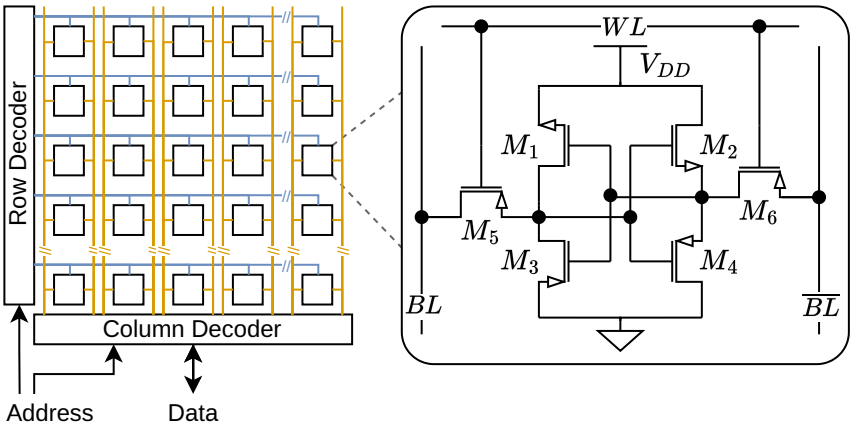


Figure 2.4: SRAM array composed of 6-transistor cells. The cells can be read by activating the corresponding word line, triggering transistors M_5 and M_6, and connecting the cell to its corresponding bit lines.

713× and energy benefits of 1855× when compared with a conventional CPU (Intel Xeon Gold 5118). Furthermore, when compared with an NVIDIA® GeForce RTX 3080 Ti GPU, pLUTo still demonstrates a 1.2× performance improvement and a reduction of 39.5× in energy consumption. However, pLUTo requires redesigning the actual memory device in order to accommodate its low-level computing features while imposing a hardware overhead that can be as high as 23.1% of the die area. Furthermore, even though the LUT queries shorten the computation time, the number of these structures required for accelerating a given computation may be prohibitively high given that all possible outcomes of every PIM function have to be pre-computed and stored.

2.1.2 Static Random-Access Memory

Recently, some works suggested moving PIM from the main memory (DRAM) to the cache. These works take advantage of the Static Random-Access Memory (SRAM) technology to perform bit-line computation, (similarly to the DRAM) achieving massive parallelism due to the memory's high internal bandwidth and also taking advantage of the operands' locality.

Figure 2.4 illustrates a SRAM array composed of 6-transistor memory cells. Similarly to DRAM, SRAM is a volatile memory technology, i.e., it needs to be powered on to keep the stored information. The process of reading an SRAM cell starts with pre-charging both the bit lines (BL and $\overline{BL}$) to V_{DD}. Then, the word line (WL) is driven high, activating transistors M_5 and M_6, which will connect the memory cell to the bit lines, pulling down one of them. The resulting differential is then sensed and the value of the memory cell is read. The complementary writing process is similar to the reading process, except that when transistors M_5 and M_6 are activated, one of the bit lines is driven low externally, flipping its counterpart and setting the state of the memory cell.

When the word line is in the low state, the SRAM cell is said to be on hold.

Differently from DRAM, SRAM does not suffer from current leakage and does not need periodic refreshing, since SRAM cells are built using bistable latching circuitry that forms a flip-flop. This flip-flop retains its state (either '0' or '1') as long as power is supplied to the circuit. Because the state is maintained by the stable configuration of the transistors, there is no need for periodic refreshing to retain data. This causes voltages sensed during a read operation to fluctuate much less than in DRAM. Consequently, bit-line computation in SRAM has proven to be more robust than in DRAM, being less error-prone and allowing to successfully combine more bit-lines at once [31].

For example, Aga et al. [3] propose a cache computing architecture where the cache resources are re-purposed to perform active computation. In this work, bit-line computation is used, similarly to early PIM solutions for DRAM. Also, extensions to bit-line computation were added to enable support for additional operations, such as *copy*, *xor*, *equality comparison*, *search*, and *carryless multiplication*. In total, the devised architecture supports 9 operations, as listed in Table 2.2. The architecture is programmable using explicit routines supplied by the authors, therefore no compiler support is required. Transparently to the programmer, the architecture provides two different computing mechanisms: *in-cache* and *near-cache*. The *in-cache* computation mechanism can only take place when the operands are located in cache blocks within a certain physical region that the authors define as Block Partition (BP), illustrated in Figure 2.5. If such condition is met, standard bit-line computation can be used through the simultaneous activation of multiple rows. Otherwise, only *near-cache* computation is possible. The *near-cache* computation mechanism is composed of discrete logic and takes place at the memory controller level.

The computation *in/near-cache* can occur in any of the cache levels, depending on the locality of the operands. However, to simplify the design, the authors decided that the computation can only take place at the L1 or Last-Level Cache (LLC) (L3). Figure 2.6 illustrates a working example

Table 2.2: Supported operations of the Compute Cache devised by Aga et al. [3].

Operation	**Description**
copy	$b[i] = a[i]$
zeroing	$a[i] = 0$
equality comparison	$r[i] = (a[i] == b[i])$
search	$r[i] = (a[i] == k)$
and	$c[i] = a[i] \& b[i]$
or	$c[i] = (a[i] \| \| b[i])$
xor	$c[i] = a[i] \oplus b[i]$
carryless multiplication	$c_i = \oplus(a[i] \& b[i])$
not	$b[i] = !(a[i])$

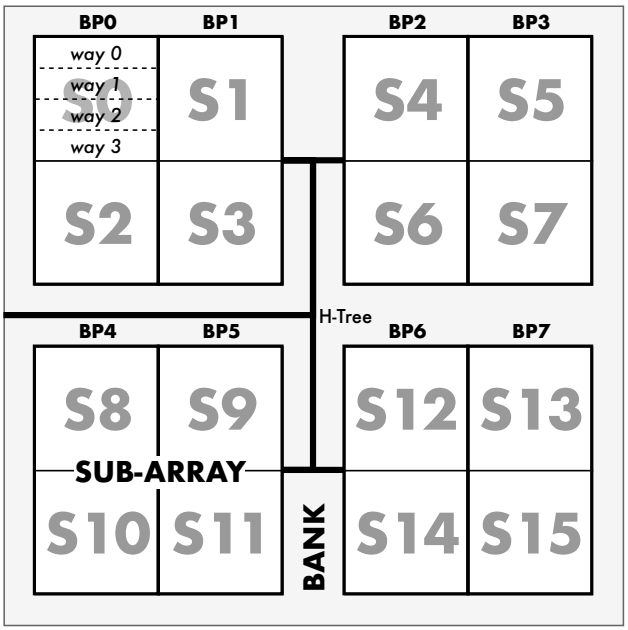


Figure 2.5: Cache hierarchy as described by Aga et al. [3]. $BP0 \dots BP7$ identify different Block Partition, whereas $S0 \dots S15$ represent different sets. *In-cache* computation can only take place when the operands are located in the same BP, otherwise only *near-cache* computation is possible.

of the Compute Cache where the computation takes place at the L3. First, the core issues a Compute Cache operation to the L1 controller ①. As none of the operands is present in the L1 cache, the instruction is forwarded to the L2 ②. The L2 cache has a dirty copy of B and a clean copy of A, however, the computation cannot take place at L2, so B is written back to the L3 and the instruction is forwarded down ③. The resulting address, C, is not cached in the L3, so it is fetched from memory ④, the operation is performed ⑤, and the L1 controller is notified of its completion ⑥. Finally, the L1 controller notifies the core that the instruction has been executed ⑦.

The system was modeled using an architectural simulator (SniperSim [51]) and a power, area, and timing modeling framework (McPAT [52]) was used to model energy consumption in the cores and caches. The base system that was used for comparison supported 32-byte SIMD loads and stores. Results for a set of micro-benchmarks show that whereas the modifications to the cache result in an area overhead of 8%, the achieved speedup can be as high as $54\times$, with an average energy saving of 91%. Complementary to these microbenchmarks, a set of applications was adapted to use the Compute Cache. Performance improvements ranging from $1.5\times$ to $3.2\times$ where achieved and energy efficiency improvements of $2.7\times$ (on average) were observed.

Another implementation was presented by Subramaniyan et al. [6], who proposed an application-specific architecture where the processing also takes place *in-cache*. They suggest exploiting the last-level caches to perform active computation optimized for the Nondeterministic Finite Au-

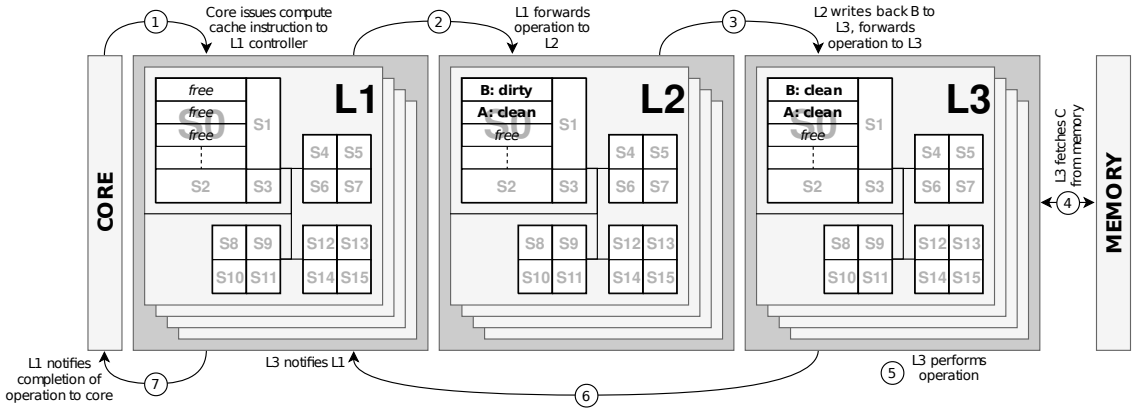


Figure 2.6: Operation example of the Compute Cache devised by Aga et al. [3] where the computation takes place at the LLC.

tomaton (NFA), which is a computational model widely used in data analytics, data mining, and network security, among others. Differently from traditional PIM approaches, including Micron's DRAM-based automata processor [53], the Cache Automaton relies on cutting-edge SRAM cache technology, which provides faster and more energy-efficient memories. The switch architecture, responsible for coordinating the access to the common buses, is redesigned to support the automata processing features, as well as the line multiplexing for sense-amplifying. Compiler support is also provided, which automates the process of mapping real-world NFAs to the Cache Automaton. The authors provide two different architectures, one optimized for performance and the other for low cache footprint. Their results show that the system optimized for performance provides a speedup of 15× over Micron's automata processor and 3840× over a conventional x86 CPU core, while the architecture optimized for low cache footprint provides a speedup of 9× over Micron's automata processor, and reduces the cache utilization by $40\,\%$, on average. However, the authors support the evaluation of their system on a GPU simulator, which they repurposed to simulate the behavior of the proposed Cache Automaton. Their choice of assessment platform, together with the lack of details regarding the adaptations made to the simulator to support the simulation of NDP devices, puts into question the accuracy of the presented results, especially given their discrepancy when compared with commercial high-performance processing devices.

2.1.3 Resistive Random-Access Memory

RRAM bit-cells, also called memristors, are devices with two terminals composed by metal electrodes and a switching oxide stack [54]. The application of a certain programming voltage between the two electrodes changes the conductivity of the metal oxide, which leads to a switch between two stable resistance states: Low-Resistance State (LRS) and High-Resistance State

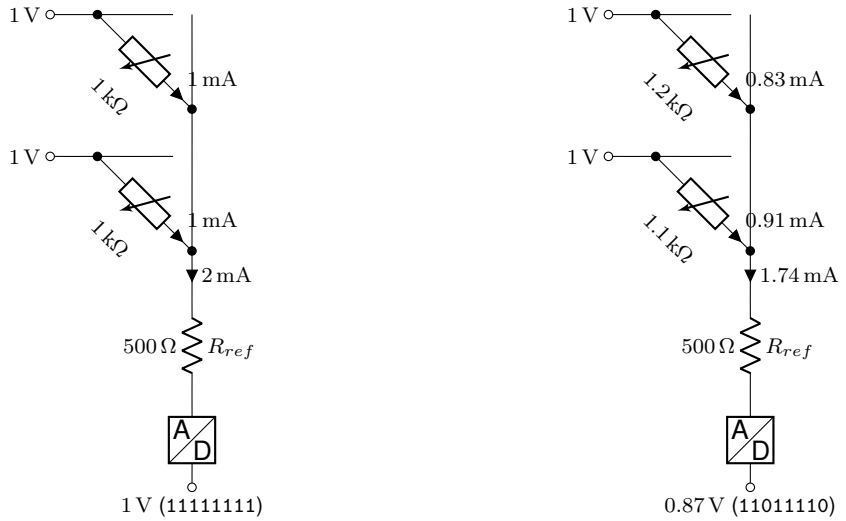


Figure 2.7: Operation principle of the RRAM-powered analog dot product: (a) under perfect conditions; (b) in the presence of variations. When subjected to variations, the final sum of the currents flowing to the memristors has an associated error which is reflected in the output of the 8-bit Analog-Digital Converter (ADC).

(HRS). Thus, allowing to encode the binary values '1' and '0', respectively. The switching from HRS to LRS is called a set process and can be achieved through the application of a positive programming voltage. The complementary process is called reset process and involves the use of a negative programming voltage.

Although classic applications of memristors only use two possible states to encode binary data, the physical characteristics of memristors allow the LRS to assume a wide range of values. This particular feature allows to use RRAM cells to store non-binary values (encoded as physical impedance), which can be used to implement hardware analog dot-products [7, 55–57]. To do that, the RRAM cells are organized into a crossbar of passive arrays, removing all the access transistors. The output current becomes the sum of the currents flowing through each memristor, which has encoded a certain non-binary value in the form of impedance. In other words, the output current is equal to the sum of the input voltages weighted by the values encoded in the memristors (see Figure 2.7a).

In theory, due to the high density of RRAMs, the use of RRAM-based analog computation enables massive parallelism leading to remarkable performance and energy efficiency benefits [7]. However, RRAM-based analog computation suffers from two limitations that largely reduce its applicability. First, it requires a massive quantity of Digital-Analog Converters (DACs) and ADCs, which are area- and power-hungry, limiting the scalability of RRAM-based analog computing devices. Second, the physical characteristics of memristors may vary due to process variations dur-

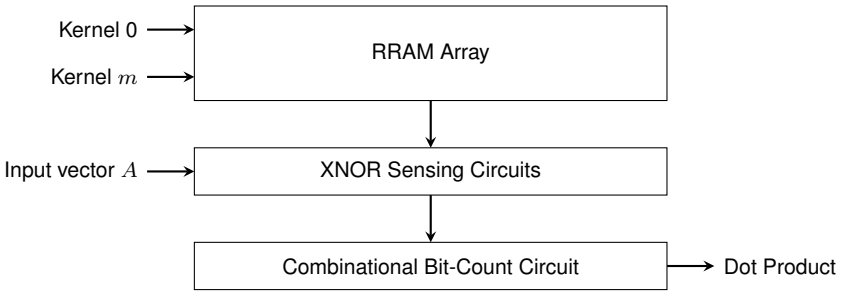


Figure 2.8: Simplified structure and operation of the RRAM-based binary convolution block proposed in [58].

ing fabrication as well as temperature oscillations during the device operation [8]. This severely decreases the reliability of the analog capabilities of RRAMs, as the results are severely affected by the lack of precision of their impedance values. For example, Figure 2.7b illustrates how a $15\,\%$ variation in the impedance of two memristors (on average) introduces an error in the sensed digital result of an analog multiplication.

The works presented in [11, 58, 59], which also use RRAM arrays to perform computation, mitigate this effect by allowing only two levels of impedance, thus implementing RRAM-based digital processing. Naturally, this approach is much more tolerant to the impedance variations of memristors, allowing to obtain reliable results. In particular, in [59] the authors propose a convolutional block to accelerate the inference phase of Binary Neural Networks (BNNs) by executing convolutional layers directly in the RRAM using a convenient fully-digital approach.

Due to the nature of the operands in a binary convolution (each element of the operands has only one bit), the costly operations of multiplication and accumulation required by regular convolutions can be replaced by *xnor* and bit-count, respectively [60], which can be efficiently implemented in hardware using the low-level properties of memristors and some additional circuitry. Note that in the context of Neural Networks (NNs), convolutions are implemented as a sequence of several dot products. Therefore, the block that implements the binary convolution executes a sequence of vectorial *xnor* operations, one per cycle, until producing the final result of the convolution.

Figure 2.8 shows the simplified structure of the convolutional block proposed in [59]. The sequences of binary weights (kernels) are stored in the RRAM array, one per row. To perform the dot product between a kernel and an input vector, the line of the RRAM containing the kernel is selected and the input vector is applied to the RRAM array. Then, the bit-wise *xnor* between the operands is performed as a readout of the RRAM array, and the result is passed to the bit-count circuit, which outputs the result of the dot product operation [61].

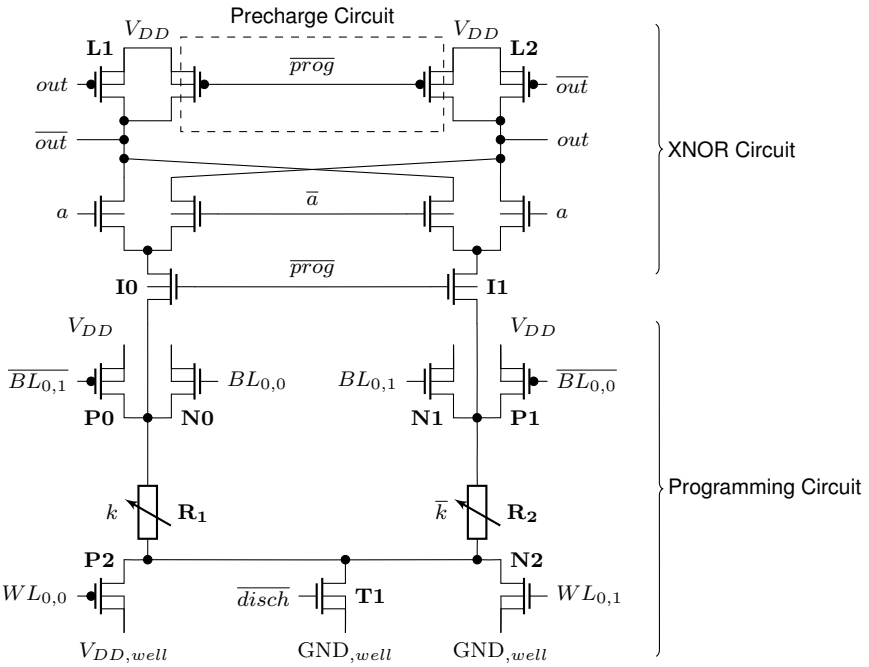
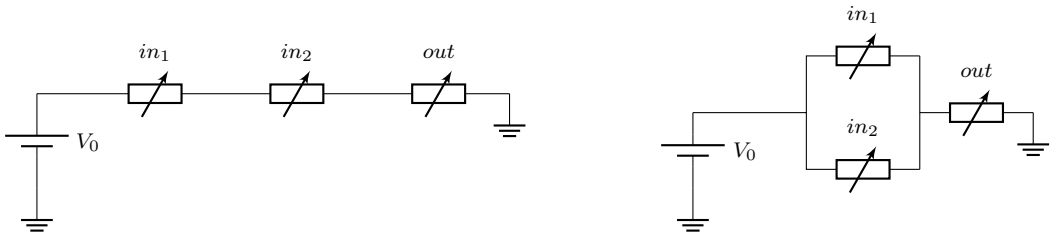


Figure 2.9: One-bit RRAM-based *xnor* cell that serves as unit of the RRAM array used in the convolution block as illustrated in [58]. Signal a represents a single element of the binary input vector A, and k represents the corresponding element of the selected kernel.

In particular, Figure 2.9 illustrates the portion of the convolution block specifically responsible for implementing the *xnor* operation, which is composed of three parts: the programming circuit; the precharge circuit; and the *xnor* circuit. The programming circuit is responsible for storing the kernels in the RRAM array. The precharge circuit loads the outputs (out and $\overline{out}$) to V_{DD} before the readout. Finally, the *xnor* circuit is responsible for performing the *xnor* operation between the input and the kernel stored in the selected row of the RRAM.

Accordingly, the operation of the convolutional block can be divided into two phases: (1) the kernel storage phase; and (2) the computing phase. During the kernel storage phase ($prog = 1$), both the outputs, out and $\overline{out}$, are precharged to V_{DD} and the RRAM array is isolated from the *xnor* circuit through transistors $I0$ and $I1$. Additionally, $\overline{disch}$ is set to GND, turning $T1$ off. The value of k (an element of the selected kernel) is stored in the RRAM in a complementary way. For example, when $k = 0$, R_1 is set to HRS and R_2 to LRS. R_1 is set to HRS by turning on transistors $N0$ and $P2$ and turning off transistors $P0$, $P1$, $N1$, and $N2$. During the reset process of R_1, $V_{DD,well}$ and $GND_{,well}$ are switched to $2 \times V_{DD}$ and V_{DD}, respectively. As such, the voltage difference across R_1 becomes $-V_{prog} = -2 \times V_{DD}$, which triggers a reset process. Similarly, R_2 is set to LRS by turning on transistors $P1$ and $N2$ and turning off transistors $P0$, $P2$, $N0$, and $N1$.

The computing phase consists of calculating the *xnor* between a (an element of the input



(a) MAGIC *and* gate. (b) MAGIC *or* gate.

Figure 2.10: Example showing two-input MAGIC (a) *and* and (b) *or* gates.

vector A) and k (an element of the selected kernel) stored in the memory. This phase is performed as a single memory readout thanks to the *xnor* sense amplifiers. The read sequence goes as follows: first, $V_{DD,well}$ and $GND_{,well}$ are switched to V_{DD} and GND respectively. Signal $prog$ is set to '0' and $\overline{disch}$ is set '1'. Hence, nodes out and $\overline{out}$ are grounded through the memristors and transistor $T1$. During this phase, the complementary states of the memristors control the discharge currents, so that the memristor in LRS is discharged faster than the one in HRS. When out or $\overline{out}$ reach the voltage $V_{DD} - V_{T,pmos}$ ($V_{T,pmos}$ denotes the threshold voltage of transistors $L1$ and $L2$), the associated *pmos* transistor is turned on, pulling up the other node (out or $\overline{out}$) to V_{DD}. The resulting output is the *xnor* of a and k.

To evaluate the proposed RRAM processing technique, the authors have evaluated their system and compared the results with those obtained with a high-performance ARM Cortex-A53, showing performance improvements of $11.3\,\%$ and energy savings of $7.3\,\%$, at the cost of only $0.3\,\%$ additional chip area.

Using a different approach, Kvatinsky et al. [10] propose MAGIC–Memristor-Aided Logic. The proposed technique utilizes memristors to store and process data simultaneously, performing computations directly within memory arrays. MAGIC employs a crossbar array structure where memristors are placed at intersections of horizontal and vertical wires, serving as the computational fabric. Data is represented by the resistance states of memristors, with HRS representing logic '0' and LRS representing logic '1'.

MAGIC can perform logic operations by exploiting the memristors' physical properties and the crossbar array structure. Figure 2.10 illustrates two-input MAGIC *and* and *or* gates. The operation of a MAGIC gate involves two sequential stages. In the first stage, the output memristor is initialized to a known logical state. During the second stage, a voltage V_0 is applied across the logic gate. The voltage across the output memristor during this phase depends on the logical states of the input and output memristors. The nonlinear characteristics of the memristor, specifically the threshold currents or voltages, are used to ensure proper operation. For certain

input combinations, the voltage is sufficient to change the logical state of the output memristor, meaning the memristor voltage or current exceeds the threshold. For other input combinations, the output remains in its initialized state, with the memristor voltage or current staying below the threshold. For instance, an *and* operation between two input vectors is executed by applying the vectors to the horizontal and vertical wires, resulting in low resistance at intersections where both inputs are '1'. The memristors' ability to maintain resistance states enables stateful logic, allowing computation results to be stored and reused without data transfer.

The crossbar array structure inherently provides massive parallelism, enabling multiple operations to occur concurrently. Additionally, MAGIC also supports analog computation by leveraging memristors' continuous resistance states, facilitating efficient implementations of operations like vector-matrix multiplication and NN computations.

While MAGIC demonstrates the potential of memristor-based in-memory computing, reducing data movement and enabling parallel processing, practical implementations face challenges such as memristor variability, sneak path currents, and integrating memristor arrays with conventional computing systems. Nonetheless, MAGIC offers a promising approach for leveraging memristors' unique capabilities to enable efficient computation within memory, potentially improving performance and energy efficiency for data-intensive applications. Several recent works use the mechanisms proposed by MAGIC to enable NDP using RRAM [62–66].

2.2 Processing Near Memory

While PIM is capable of exploiting the high bandwidth to memory (by performing computation inside the memory chips) and attaining massive parallelism, it is also severely constrained by the preexisting memory architecture, limiting the design exploration of in-memory computing architectures. Furthermore, most traditional PIM architectures can only implement simple logic operations (often needing thousands of memory cycles to perform more complex arithmetic instructions) and require the programmer to be aware of the underlying memory architecture to use such mechanisms.

On the other hand, by placing the functional units right outside the memory controller, PNM allows for a much wider design exploration space, with the possibility of engineering fully digital processing circuitry next to memory devices. As a consequence, PNM architectures are not as dependent on the physical placement of data in memory as PIM devices, and specific knowledge about the underlying architecture of memory devices is not required.

For example, UPMEM [67] specializes in developing PNM architectures that utilize Data Pro-

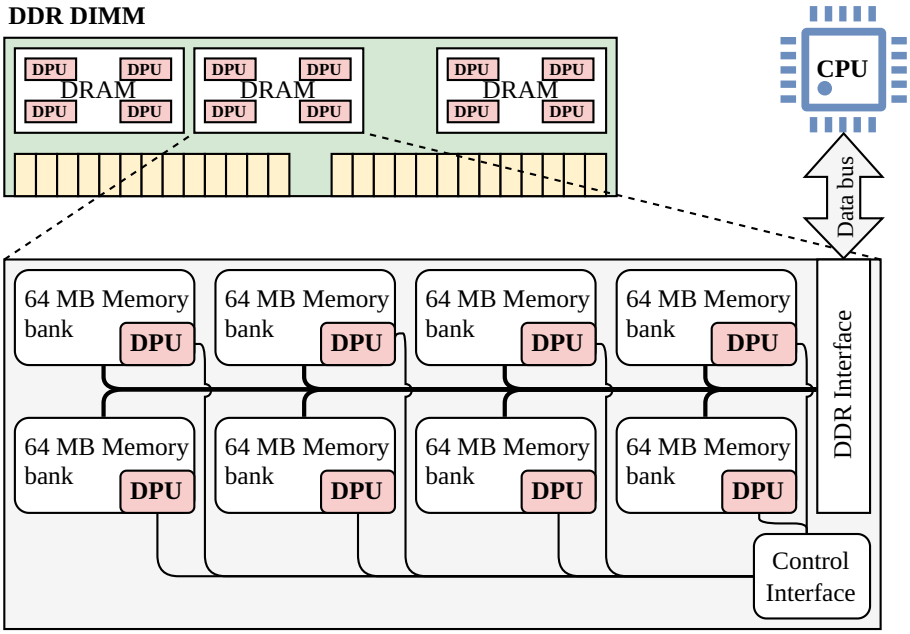
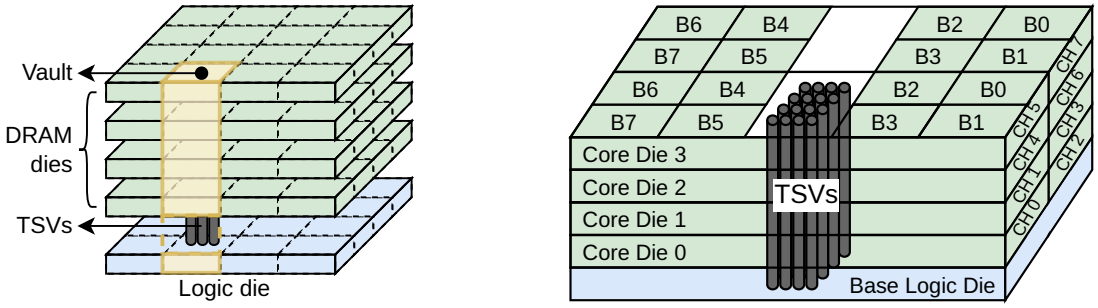


Figure 2.11: Overview of UPMEM's PNM-enabled memory devices. Each DRAM bank contains a DPU consisting of a simple general-purpose in-order CPU for processing data.

cessing Units (DPUs) inside DRAM banks to process data, as illustrated in Figure 2.11. Each DPU consists of a simple general-purpose in-order CPU with 11 pipeline stages, 24 interleaved hardware threads, and 32 registers per thread. UPMEM's scalable and programmable PNM solution lower the total cost of ownership for data centers by 5× to 10× compared to traditional Field-Programmable Gate Array (FPGA)- or GPU-based solutions [68–70]. The adopted approach is particularly beneficial for big data, Artificial Intelligence (AI), genomics, and database applications, with several recent proposals using UPMEM's NDP architecture [71–81].

However, UPMEM's NDP-enabled memory devices are specialized for performing NDP, and may slow down classic non-NDP workloads. This effect can be explained by the additional traffic generated by data being moved between DRAM banks. Since DPUs are installed inside each DRAM bank, the data to be processed needs to be contained inside a specific bank, otherwise, the operands must be explicitly transferred to a specific bank before processing, increasing traffic and (possibly) generating contention. Naturally, this also means that the applications must be re-designed to harvest the performance benefits enabled by this NDP solution, and the programmer needs to be aware of the physical architecture of UPMEM's memory device. Furthermore, developing NDP architectures such as UPMEM's requires re-engineering and re-fabricating DRAM chips, which is a complex and expensive process.

In fact, until the past decade, there were virtually no mechanisms available to install NDP devices next to memory arrays inside the same chip without redesigning the entire circuit. Hence,



(a) Hybrid Memory Cube architecture.

(b) High-Bandwidth Memory architecture.

Figure 2.12: Memory technologies including both 3D-stacked DRAM and logic dies, enabling the implementation of PNM architectures within the same chip as memory arrays.

the only alternative was to implement PNM devices in a separate chip and solder it in the same logic boards as memory chips. While still being able to achieve higher bandwidths to memory than the CPU, inter-chip buses compare poorly with intra-chip communication lanes both in terms of energy efficiency and performance by several orders of magnitude [82, 83].

It was not until 2011 when Micron Technologies introduced 3D-stacking DRAM arrays and logic layers that memory devices and PNM architectures could finally co-exist within the same chip without requiring to redesign it entirely [84–86].

2.2.1 Hybrid Memory Cube and High-Bandwidth Memory

The Hybrid Memory Cube (HMC) [87] and the High-Bandwidth Memory (HBM) [88] are two high-performance DRAM interfaces that have been engineered to deliver greater bandwidth than traditional memory configurations. HMC, illustrated in Figure 2.12a, distinguishes itself with a 3D-stacked memory architecture and an abstracted cube design which enables more advanced transaction and protocol execution. It comprises multiple layers of DRAM cells stacked atop a base logic layer and utilizes high-speed Through-Silicon Vias (TSVs) to interface with the memory controller that resides in the logic die.

On the other hand, HBM (see Figure 2.12b) also employs 3D stacking and TSVs but usually features fewer layers of DRAM. It can be more closely integrated with the surrounding processing devices, such as CPUs, GPUs, or NDAccs, potentially residing on the same interposer or chip, which allows it to achieve remarkable transfer speeds. This close integration is particularly beneficial for NDAccs, which are capable of exploiting the massive bandwidth allowed by the HBM [89–91].

While the HMC offers a high bandwidth per pin and is targeted at high-performance computing and networking applications, it hasn't seen broad adoption in the consumer market, remaining

somewhat specialized. In particular, deploying NDAccs into the HMC may be challenging since the logic layer is divided into small sections connected to different portions of the DRAM arrays called vaults, and an NDAcc not only has to be contained into a single section but also can only communicate with the portion of the DRAM arrays within the same vault. In contrast, the logic die of HBM is not partitioned, while still offering massive bandwidths through on-chip memory TSVs.

Two examples of NDP-enabled devices based on HBM that have been proposed and integrated with commercial processing systems are Aquabolt-XL [89] and AXDIMM [92]. The Samsung Aquabolt-XL places a cache-line-width Single Instruction Multiple Data (SIMD) PNM computation unit at each bank of the HBM. By using a cache-line-sized data access granularity consistent with that of the CPU, Aquabolt-XL solves many issues naturally arising in PIM architectures, such as data layout, and virtual memory management [90]. Furthermore, since this device is programmed and controlled through extended DRAM commands from existing HBM controllers, it can be plugged into standard systems. In addition, Samsung also provides PNM device drivers, Application Programming Interfaces (APIs) for TensorFlow and PyTorch, and libraries to enable PNM. By exploiting significant parallelism at bank-level, Aquabolt-XL can deliver up to 2× speedup while reducing energy consumption by more than $70\,\%$. On the other hand, AXDIMM utilizes in-DIMM data processing units to enable NDP at rank-level[1], localizing data movement. Similarly to Aquabolt-XL, AXDIMM can also be plugged into standard processing systems, being programmed and controlled by software extensions. AXDIMM is shown to provide over $80\,\%$ performance improvements while reducing energy consumption by more than $40\,\%$ in a two-rank configuration. In addition, the presented experiments indicate that the performance benefits offered by AXDIMM scale almost linearly with the number of ranks, with 3.5× and 6.9× speedups having been observed to 4-rank and 8-rank configurations [90].

2.3 Multi-Level NDP Architectures

In addition to traditional (single-level) NDP architectures, multi-level NDP architectures have also been proposed in the past years, which are capable of offloading NDP-enabled workloads to several memory levels, depending on what level offers the most benefits.

For example, Ahn *et al.* [43] have introduced the concept of PIM-Enabled-Instruction (PEI), which are instructions that are in the format to be executed by an existing NDP mechanism, but they are not necessarily issued to the memory. Their work also relies on architectures with previous NDP support, therefore requiring minimal modifications to the compiler and the programming

[1] Set of DRAM chips accessed in parallel to provide the full data word width required by the memory controller.

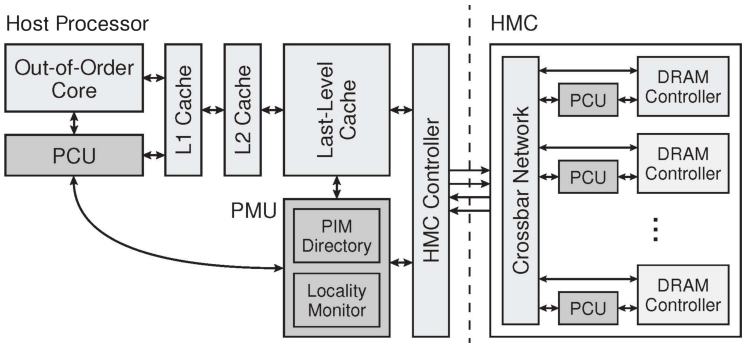
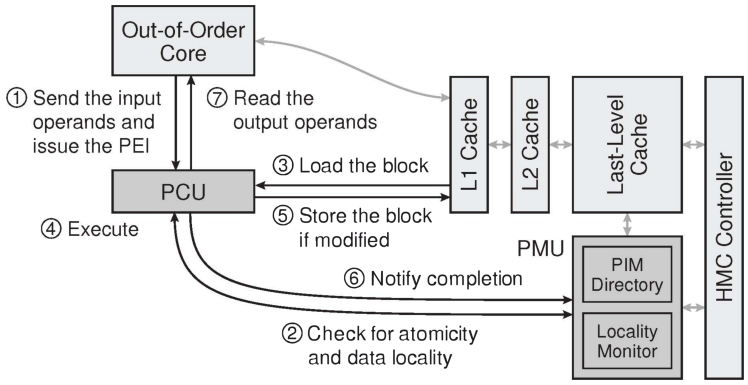


Figure 2.13: Overview of the system proposed by Ahn et al. [43]. The dark gray blocks identify the devised hardware modules for processing PEIs (PCUs) and for ensuring atomicity in the execution of the instructions as well as deciding where the PEIs should be processed (from [43]).

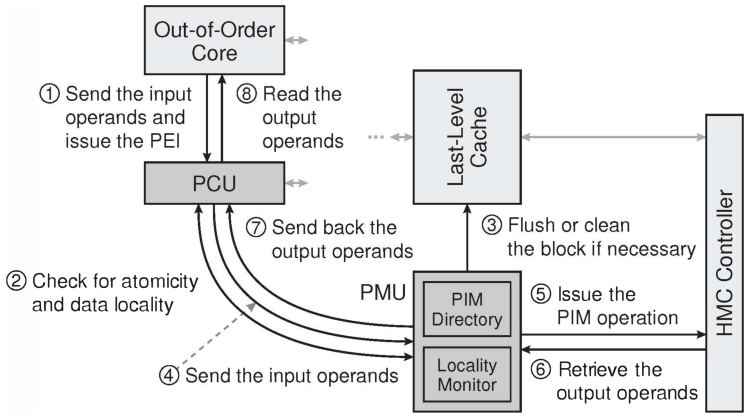
paradigm. The core idea is to create an equivalent PEI for each supported NDP instruction and replace the original instruction by the new one. When a PEI is issued by the host processor, there is a hardware mechanism that determines the best place to process it between memory and host processor. The software is unaware of where the instruction is processed, which is entirely managed by hardware units. The devised mechanism is mainly composed of two types of hardware units: (1) The PEI Management Unit (PMU) and (2) the PEI Computation Units (PCUs). The PMU aggregates the main control logic of the system, and has the responsibilities of serializing the execution of PEIs (PIM Directory) and decide whether PEIs should be executed in-memory or in the host processor, by monitoring the locality of the operands (Locality Monitor). The PMUs are responsible for executing PEIs and are installed in the memory chip (for in-memory processing) and near the processing cores, as shown in Figure 2.13.

Depending on the locality of the operands (which is checked by the locality monitor inside the PMU), the PEI can be executed near the processing core (high data locality) or directly in memory (low data locality). Figures 2.14a and 2.14b show the PEIs execution flow when they are issued to the PCU near the processing core or the PCUs inside the memory device, respectively.

When a PEI has high data locality, the PMU will assign its execution to the PCU near to the host processing core (Figure 2.14a). First, the CPU sends the operands to the PCU and issues the instruction ①. Secondly, the host processor accesses the PIM Directory inside the PMU to obtain the lock of the cache block that contains the operands and consults the Locality Monitor to decide whether the PEI should be processed near the host processor or in-memory ②. If the operands are closer to the processing core (in cache), the PMU advises the execution on the host-side PCU. Therefore, the host-side PCU allocates an operand buffer entry to store the operands, loads the cache block from the L1 cache ③ and executes the PEI ④. Meanwhile, if the PEI modifies the target cache block, a store request is issued ⑤. When the PEI ceases to



(a) Host-side PEI execution



(b) Memory-side PEI execution

Figure 2.14: Auxiliary diagrams to explain the execution of PEIs both in the processor-side and the memory-side (from [43]).

execute, the PCU notifies both the host processor and the PMU, which maintains the consistency and ensures that the results of the PEIs are committed in the same order the PEIs where issued ⑥. Finally, the processor accesses the output operands through the memory-mapped registers of the PCU and releases the buffer entry.

If the operands of a PEI are not present in cache, the PMU issues the execution to the memory device (Figure 2.14b). Steps ① and ② are analogous to the case above. However, in this case, the PMU informs the host-side PCU that the PEI is to be processed on the memory-side and sends a back-flush to the LLC to write-back any dirty data that the block may contain and invalidate its content for the next reading request ③. Also, at the same time, the host-side PCU sends the input operands stored in its mapped registers to the PMU ④. The operation type, the target block addresses, and the input operands are used to build a packet, following the memory device protocol for NDP operations, which is sent to the main memory ⑤. Upon completion, the memory device sends back a response packet ⑥, the PMU is notified and sends the output operands to the host-side PCU owned by the core that issued the PEI ⑦, and finally, the processor reads them

through the PCU memory-mapped registers ⑧.

To solve data consistency and address translation problems, the authors impose an essential *single-cache-block-restriction*, which dictates that a PEI can only access one single LLC cache block. According to the authors, this brings several advantages such as simplified locality profiling and address translation and essential implications at data localization and interoperability levels. However, it also limits the scope of operation, since the data has to be placed following a particular strategy to be processed by the devised system.

The authors designed this architecture to target applications with large memory footprints and large memory bandwidth consumption, often classified as big data workloads. A set of applications from areas such as large-scale graph processing, in-memory data analysis, machine learning, and data mining was compiled to test the system, rather than a standard battery of benchmarks. Therefore, the instructions implemented by the PCUs are useful for the considered applications. Table 2.3 presents the instructions supported by the PCUs as also the applications that benefit from each instruction.

For large inputs, it was observed that the conventional NDP-only system achieves 44% speedup over the ideal host processor since massive parallelism is achieved using the internal bandwidth of the memory. However, for small inputs, the NDP-only system degrades the overall performance by 20% because it always performs the computation in the DRAM even though the data is present in cache. Hence, the devised system not only maintains the speedup for large inputs, using NDP, but also competes with the host-only system for small operands. On average, the overall system outperforms both the NDP-only and the host-only systems by 32% and 47%, respectively. Concerning area and energy consumption, negligible overheads were determined for both. More

Table 2.3: Instructions implemented by the PCUs in the architecture proposed by Ahn et al. [43].

Operation	Read	Write	Input	Output	Application(s)
8-byte integer increment	Yes	Yes	0 bytes	0 bytes	Average Teenage Follower
8-byte integer min	Yes	Yes	8 bytes	0 bytes	Breadth-First Search, Single-Source Shortest Path, Weakly Connected Components
Floating-point add	Yes	Yes	8 bytes	0 bytes	PageRank
Hash table probing	Yes	No	8 bytes	9 bytes	Hash Join
Histogram bin index	Yes	No	1 byte	16 bytes	Histogram, Radix Partitioning
Euclidean distance	Yes	No	64 bytes	4 bytes	Streamcluster
Dot product	Yes	No	32 bytes	8 bytes	Support Vector Machine Recursive Feature Elimination

specifically, the memory-side PCUs contributes with only 1.4% for the energy consumption of the main memory and about 1.85% for the die logic area.

By, using a different approach, Lockerman *et al.* [93] have recently proposed Livia, a versatile NDP system that can perform computations at multiple levels of the memory hierarchy, resulting in moderate performance and energy consumption benefits (even for challenging irregular workloads) while keeping the area overhead under $3\,\%$. Livia introduces compute units integrated into the main memory, called Livia Memory Units (LMUs), and compute units integrated into the LLC, called Livia Cache Units (LCUs). These units allow data-parallel operations to be performed near the data, reducing the need for expensive data transfers between memory and the CPU. Livia also extends the CPU's Intruction Set Architecture (ISA) with new instructions to offload computation to the LMUs and LCUs. A software application manages the offloading of workloads to these units at runtime, as well as data movement and synchronization. The proposed system determines the optimal NDP-enabled memory level to offload a workload in an attempt to maximize the potential performance benefits, based on the analysis of parameters such as locality of data and availability of processing resources.

In their paper, the authors evaluate Livia using various data-intensive workloads, such as machine learning, databases, and graph analytics. The results demonstrate that Livia can provide performance improvements of up to 7.5× and energy savings of up to 5.4× compared to a baseline CPU-only system.

However, Livia does not actually propose novel NDP architectures. Instead, it employs previously existent NDP devices, and the novelty of this work relies on the hardware circuitry that determines where the workload is processed (the CPU or some level of the memory hierarchy). Moreover, Livia is limited to the architectures of the NDP solutions it employs, making it incapable of accommodating different NDAccs. In addition, the authors evaluated Livia by prototyping it in an FPGA device, which certainly does not have enough resources to implement a complex high-performance Out-of-Order (OoO) CPU and NDP-enabled memory devices, indicating that the evaluation was based on an incomplete implementation of the proposed system. Consequently, their evaluation methodology fails to account for the interactions between the different components of the system that affect the performance of the NDAccs, leading to a potentially inaccurate conclusion.

2.4 Development Technologies for Validation and Evaluation

Although PIM and PNM mechanisms are already adopted on multiple devices available in the market supporting either paradigm, and memory technologies supporting the implementation of PNM engines close to memory arrays are already available, the supporting technologies for developing and testing new NDP architectures are still scarce. The following sections provide an overview of the existing NDP development mechanisms, pointing out the strengths and shortcomings of each solution.

2.4.1 Classic RTL Development Technologies

The design of novel NDAccs is a complex task. Moreover, directly prototyping such accelerators by developing Register Transfer Level (RTL) implementations is generally complex, expensive, and time-consuming, in particular when the intended goal is to simply evaluate the potential of an idea, or to adjust the parameters of an architecture still under consideration. In addition, without properly integrating the accelerator with the remaining system, particularly with the main CPU and the memory hierarchy, such methodologies may lead to inaccurate results. These inaccuracies are mainly due to control and synchronization overheads or to the contention generated at the memory level attributed to the co-existence of several processing devices simultaneously accessing the memory hierarchy.

For example, PiMulator [94] proposes an FPGA-based approach where the authors provide a soft-hardware infrastructure, including a soft-CPU core, making it easier to implement custom circuitry that emulates PIM. Their tool is compatible with the LiteX System on Chip (SoC) framework [95] and supports a representative set of PIM architectures. However, PiMulator has severe limitations imposed by the resources available in FPGA devices, which do not allow to accurately emulate complex hardware structures, such as superscalar OoO CPUs or the internal architecture of a DRAM array. Hence, it is unlikely that such a tool can accurately reproduce the overheads generated by control, synchronization, and memory contention that are present in a real system. Furthermore, developing new NDAccs still poses a significant implementation effort, since the devices have to be described using Hardware Description Languages (HDLs).

On the other hand, development technologies supported by simulation are an easier approach, as pure simulation environments are not as constrained by physical hardware resources, usually allow for much faster development and validation of architectures, and have no fabrication cost. Furthermore, such solutions support much higher-level modeling paradigms, which are useful for validating and preliminary evaluating ideas, when the architecture of the NDP devices under

development are not yet fully parameterized.

2.4.2 The gem5 Architectural Simulator

gem5 [21] is a widely-used open-source architectural simulator for computer system architectures. It is a modular and highly configurable platform that allows researchers and developers to model and simulate various components of a computer system, including CPUs, memory systems, interconnects, and peripheral devices.

gem5 supports detailed architectural models for several CPU architectures, such as ARM, x86, RISC-V, Power, and MIPS. It can simulate both in-order and out-of-order CPU pipelines, as well as multi-core and heterogeneous systems. The simulator provides a comprehensive memory system model that includes caches, main memory, and various memory controllers. It supports different cache coherence protocols and memory technologies, such as DDR, HBM, and Non-Volatile Memories (NVMs).

In addition to CPU and memory models, gem5 also includes models for multiple interconnect technologies, enabling the simulation of complex on-chip and off-chip communication infrastructures. These interconnect models cover buses, point-to-point links, and Networks-on-Chip (NoCs). Furthermore, gem5 offers models for a wide range of peripheral devices, such as disks, network interfaces, and various I/O devices, allowing to simulate complete systems.

One of the strengths of gem5 is its ability to perform both Full System (FS) and System Emulation (SE) simulation. FS simulation involves simulating the entire software stack, including the Operating System (OS), while SE simulation focuses on the execution of individual applications or benchmarks. In addition, the QEMU virtualization tool is also supported to speed up the simulation process by using the host's physical hardware to emulate part of the simulated processing system.

gem5 is designed to be highly extensible, allowing researchers and developers to modify existing models or add new models to suit their specific needs. It is written in C++ and provides a Python-based configuration and scripting interface for easy setup and automation. This extensibility has contributed to the widespread adoption of gem5 in both academia and industry, as it enables researchers to explore and evaluate novel architectural ideas and designs.

To aid in understanding and analyzing the simulated system's behavior, gem5 includes various visualization and analysis tools, such as graphical debuggers, trace viewers, and performance profilers. These tools provide valuable insights into the performance characteristics, bottlenecks, and potential areas for optimization within the simulated system.

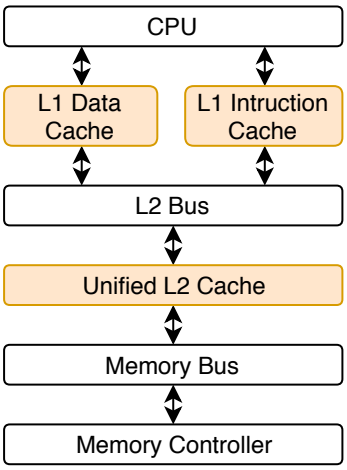


Figure 2.15: Block diagram of a processing system modeled in gem5.

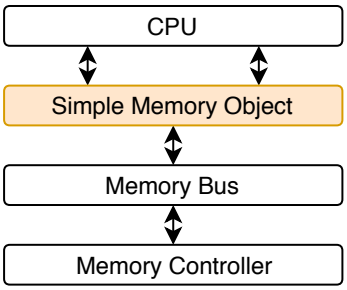


Figure 2.16: Block diagram of a system featuring a CPU and a main memory connected by a simple memory object whose only goal is to forward the requests between the two entities.

2.4.2.A Simulating Processing Systems with gem5

When using gem5, there are three main types of files: (1) source files; (2) object files; (3) and script files. The source files contain the behavioral description of a component's architecture and are written in C++. They all inherit from the supper class *SimObject* and implement a set of methods required for the architecture to be simulated. Each time the architecture of a simulation object is changed, it has to be recompiled. On the other hand, the object files are Python classes that declare the interface exported by a simulation object. Optionally, it can pass parameters to the object's constructor to change dynamically the architecture (does not require recompilation). At the top level, the script files are also written in Python, and they are responsible for putting the simulation system together, by connecting the several components, specifying the configuration parameters, and also specifying the program to be executed–when the system features a CPU. The most top-level system, described in the script file, can be easily translated into a block diagram. For instance, the code in the Listing 2.1, is equivalent to the block diagram shown in Figure 2.15.

At the implementation level, each of the blocks presented in Figure 2.15 exports ports. These

Listing 2.1: Python code of a gem5 simulation script that implements a system featuring a CPU, two cache levels, and a main memory.

```
...

# Create a simple CPU
system.cpu = TimingSimpleCPU()

# Create an L1 instruction and data cache
system.cpu.icache = L1ICache(opts)
system.cpu.dcache = L1DCache(opts)

# Connect the both L1 caches to the CPU
system.cpu.icache.connectCPU(system.cpu)
system.cpu.dcache.connectCPU(system.cpu)

# Create a memory bus (a coherent crossbar)
system.l2bus = L2XBar()

# Hook the L1 caches up to the l2bus
system.cpu.icache.connectBus(system.l2bus)
system.cpu.dcache.connectBus(system.l2bus)

# Create L2 cache and connect it to the l2bus
system.l2cache = L2Cache(opts)
system.l2cache.connectCPUSideBus(system.l2bus)

# Create a memory bus
system.membus = SystemXBar()

# Connect the L2 cache to the membus
system.l2cache.connectMemSideBus(system.membus)

...

# Connect the system up to the membus
system.system_port = system.membus.slave

# Create a DDR3 memory controller
system.mem_ctrl = DDR3_1600_8x8()
system.mem_ctrl.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.master

...
```

ports allow the components to communicate using packets, which are structures defined in the superclasses from which every simulation object inherits. Packets can be requests or responses and can also contain data (not necessarily a single word).

Considering the example system depicted in Figure 2.16, the Simple Memory Object (which consists of a basic bypass that simply forwards requests) features three interfaces, two to connect to the CPU's data and instruction cache ports and one to connect to the Memory Bus. In general, the object may receive requests from both sides: the CPU side (read and write requests); and

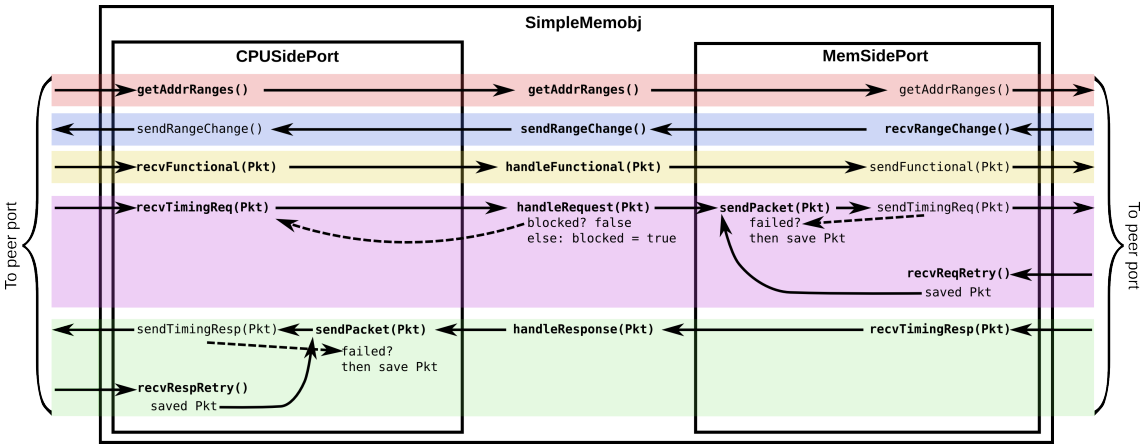


Figure 2.17: Main methods implemented by the Simple Memory Object that allow it to simply forward the packets from one side (memory or CPU) to the other. Image from gem5's official documentation available at `https://www.gem5.org/documentation/learning_gem5/`.

the memory side (when data is retrieved). To enable this, gem5 requires some routines to be implemented to deal with these requests and produce responses. Figure 2.17 shows the interface exported by the Simple Memory Object that allows it to be used as a memory component. Methods identified in bold are required to be implemented by the developed module.

In addition, gem5 provides a comprehensive set of statistics, allowing to gain deep insights into the behavior and performance characteristics of simulated computer systems. The simulator collects a wide range of statistics, including CPU metrics such as Instructions Per Cycle (IPC), branch prediction accuracy, and stall cycles; cache statistics like hit rates, miss rates, and access counts; memory system statistics like read/write counts and average access latency; and network statistics for interconnection networks in multi-core or multi-node systems. Additionally, gem5 offers the ability to estimate power consumption and energy usage based on activity statistics and power models (using tools such as McPAT [52] and CACTI [96]), enabling exploration of power-efficient designs. Users can also define and collect custom event-based statistics, providing flexibility to track specific events of interest. These statistics can be gathered at various granularities, such as per-core, per-cache, or system-wide, and can be outputted in various formats for further analysis and visualization. The rich set of statistics available in gem5 empowers researchers and architects to thoroughly evaluate and optimize the performance, energy efficiency, and overall design trade-offs of their proposed computer architectures.

However, using gem5 for pure hardware space exploration still poses some challenges. For example, while gem5 offers the capability to simulate a wide range of components right out of the box, developing new components requires implementing comprehensive internal protocols for communication between modules. While this task is relatively straightforward for the considered

Simple Memory Object (see Figure 2.17), implementing these protocols becomes more challenging for complex devices. This process is especially challenging for NDAccs because they must be tightly integrated with an existing CPU and memory hierarchy, which usually results in the necessity of adding several dedicated ports, each one with its own complex Finite State Machine (FSM). In addition, gem5 does not offer standard mechanisms to implement control and synchronization between the CPU and NDAccs.

2.4.3 ZSim Architectural Simulator

ZSim [97] is an open-source, parallel, x86-64 architectural simulator designed for modeling and evaluating the performance of modern multi-core systems. It is maintained by researchers at MIT and focuses on providing high simulation speeds, scalability, and accuracy. ZSim achieves its high performance by leveraging two key techniques: parallel simulation and fast-forwarding.

Unlike gem5, ZSim features parallel simulation, enabling it to utilize multiple cores or CPUs simultaneously for simulating different parts of the system. This parallel execution significantly reduces the overall simulation time, making it possible to simulate large-scale multi-core systems efficiently. ZSim employs a novel parallel simulation methodology that partitions the simulated system into independent components, allowing them to be simulated concurrently without sacrificing accuracy.

Additionally, ZSim incorporates a fast-forwarding mechanism that skips the simulation of non-critical sections of code, such as library initialization or system boot sequences. This technique allows the simulator to quickly advance to the region of interest, further improving simulation speed and reducing the overall simulation time. Fast-forwarding is particularly beneficial when simulating applications with long initialization phases or when the focus is on specific phases of execution.

ZSim supports detailed modeling of various architectural components, including out-of-order superscalar CPUs, multi-level cache hierarchies, and on-chip interconnects. It provides cycle-accurate simulation models for these components, ensuring high fidelity in capturing the performance characteristics of the simulated system. Furthermore, ZSim supports the simulation of multi-threaded and multi-programmed workloads, enabling the evaluation of parallel applications and resource contention scenarios.

One of the notable features of ZSim is its ability to integrate with hardware performance counters. By leveraging hardware performance monitoring units, ZSim can collect performance data directly from real hardware and incorporate it into the simulation models. This integration allows for more accurate modeling of certain microarchitectural components and improves the overall

simulation accuracy.

ZSim is designed to be extensible, allowing researchers and developers to modify existing models or introduce new components and features as needed. It provides a modular architecture and a well-documented codebase, facilitating customization and experimentation with novel architectural ideas.

However, similarly to gem5, ZSim still does not offer out-of-the-box support to model and simulate NDAccs, which is worsened by the fact that its adoption by the community is not as wide as gem5's. Therefore, while ZSim's unique features make it a powerful tool for simulating multicore and heterogeneous systems, gem5 is usually preferred for architecture space exploration due to its versatility, wide adoption by the community, and past research built upon it.

2.4.4 Ramulator

Differently from the previous solutions, Ramulator [16] is a highly flexible and cycle-accurate DRAM simulator specifically designed to model and evaluate the performance of various DRAM architectures and organizations. It is an open-source tool developed by researchers at Carnegie Mellon University and the Samsung Advanced Institute of Technology. Ramulator's primary goal is to provide a realistic and accurate simulation environment for exploring and analyzing the behavior of DRAM memory systems.

One of the key strengths of Ramulator is its versatility in supporting a wide range of DRAM standards and configurations. It can simulate various DRAM technologies, including DDR3, DDR4, LPDDR3, LPDDR4, HBM, and GDDR5. Additionally, Ramulator supports different memory organizations, such as different numbers of ranks, banks, rows, and columns, allowing researchers and engineers to explore the impact of memory organization on system performance.

Ramulator's simulation accuracy is achieved through its detailed modeling of various DRAM components and timing parameters. It accurately models essential DRAM operations, including activation, precharge, refresh, and data transfer. Ramulator also considers various timing constraints, such as row-to-column delay, column-to-column delay, and row precharge time, ensuring that the simulated DRAM behavior closely matches real-world memory systems.

Another notable feature of Ramulator is its support for non-volatile memory technologies, including Phase Change Memory (PCM) and Spin-Transfer Torque RAM (STT-RAM). This capability allows researchers to explore and evaluate emerging memory technologies and their potential impact on system performance and energy efficiency.

Ramulator provides a flexible and user-friendly interface, allowing users to configure various simulation parameters and memory configurations through a simple configuration file. This ease

of use facilitates rapid prototyping and exploration of different memory system designs and configurations.

In addition to its core simulation capabilities, Ramulator offers a range of analysis tools and utilities. These include a visualization tool for visualizing DRAM command traces, a memory trace player for replaying and analyzing memory access patterns, and a bandwidth/latency calculator for estimating the theoretical performance limits of various DRAM configurations. These features can be used to estimate the performance and energy requirements of NDAccs.

Ramulator has been widely adopted by researchers and engineers in both academia and industry for various purposes, such as evaluating the performance of memory-intensive workloads, exploring novel DRAM architectures and organizations, and optimizing memory controllers and scheduling policies. Its accuracy, flexibility, and ease of use have made it a valuable tool for understanding and improving the performance of modern memory systems.

2.4.5 NDP Development Technologies Supported on Simulation

Driven by the scarcity of NDP development technologies supported on pure simulation approaches, several recent proposals aimed at extending the previously described simulators to natively support NDP.

For example, Qureshi *et al.* [18] proposed a simulation tool titled gem5-X where they modified an L1 cache to enable in-cache processing, based on the computing architecture presented in [98]. They validated their simulation model by implementing it using hardware synthesis tools, which accurately predicted the operation of the equivalent hardware. However, their NDP capabilities are limited to a single architecture that performs computation in cache, and they only support computation in the L1 data cache. Additionally, gem5-X is based on an outdated version of gem5, making it incompatible with most recent versions.

Singh *et al.* [17] also developed a simulation-based approach for NDP-enabled systems using the ZSim architectural simulator [97] and Ramulator [16]. Their method involves executing a characterization workload to generate performance statistics and memory traces, which are then used in a post-simulation step to predict the performance of an NDP-enabled system. However, this approach does not support the simultaneous operation of the CPU and NDAcc, and the performance benefits of the NDP component are only predicted after the simulation.

Yu *et al.* [99] addressed this significant limitation in the work of Singh *et al.* by proposing a solution that, while still relying on memory traces generated by Ramulator to predict the performance benefits of NDAccs, assumes the existence of multiple threads, which allows for the simultaneous operation of both the CPU and the NDAccs. Despite this improvement, their artifact is still

constrained to DRAM-based PIM, the only memory technology supported by Ramulator, and the same ISA is used for both the CPU and the NDP devices, potentially leading to sub-optimal NDP solutions.

Two other works have been recently proposed targeting the design space exploration of heterogeneous hardware accelerators that have gained particular visibility in the community: gem5-Aladdin [19] and gem5-SALAM [20] (see Table 2.4). Both these solutions adopt an High-Level Synthesis (HLS) approach, where the developer specifies a behavioral model of the accelerators using high-level C code and structures. Then, the application code is analyzed and a Dynamic Data Dependence Graph (DDDG) is built (where the vertices are LLVM Intermediate Representation (IR) instructions and the edges represent dependencies between operations). Finally, operations are automatically scheduled to the accelerators to achieve optimal resource occupancy.

Although the high-level hardware modeling enabled by these tools facilitates the quick development and evaluation of new accelerators, it also limits the complexity of their internal architectures, which is supported by the rather simple benchmarks that were used by their authors to validate these works.

Furthermore, gem5-Aladdin and gem5-SALAM involve multi-step simulation procedures, which can make their use and adoption more difficult. gem5-Aladdin even requires analytical steps, which may introduce significant errors to the results, compared with a pure-simulation approach. In addition, these two tools do not actually produce gem5 simulation objects, indicating that the actual architecture of the simulated accelerators is unknown to gem5. Hence, the production of gem5 statistics regarding the internal operation of such components is not possible.

In addition, gem5-Aladdin only supports SE mode (with a custom virtual memory scheme to enable address translation between the virtual and the physical domains), and gem5-SALAM

Table 2.4: Summary and comparison of the main features of two NDP development tools: gem5-Aladdin [19] and gem5-SALAM [20].

	gem5-Aladdin	**gem5-SALAM**
Development tier (analogy)	HLS	HLS
Simulation flow	Multi-step, analytical analysis (requires the use of multiple tools and analytical models)	Multi-step
Performance simulation	Trace-based	Clock-cycle/interval-based
Energy estimation	Custom model	McPAT/CACTI support
Area estimation	Not supported	McPAT/CACTI support
Requires rebuilding gem5	No (architecture of NDAcc unknown to gem5)	No (architecture of NDAcc unknown to gem5)
Address translation	Custom	Not supported
gem5 simulation mode	SE only	FS bare-metal/limited support to FS with SO

only supports bare-metal FS mode and has no virtual memory support. The authors claim that this limitation can be solved using pure-software solutions (device drivers), but they provide no pointers on how that can be actually achieved.

Both solutions are compatible (to some extent) with energy and area estimation mechanisms. While gem5-Aladdin includes a custom energy model, allowing to estimate the energy requirements of the accelerators under evaluation, gem5-SALAM supports energy and area analysis through native integration with McPAT/CACTI.

2.5 Summary

In this chapter, a comprehensive background and overview of the state-of-the-art in NDP architectures is provided. NDP aims to bridge the gap between high-performance CPUs and the comparatively underperforming memory hierarchies by placing computational resources, known as NDAccs, close to where data is stored.

Two main NDP paradigms are distinguished: PIM and PNM. PIM involves repurposing existing resources within memory chips to perform active computation, co-existing in the same space as the memory cells. On the other hand, PNM places NDAccs outside the memory controller or even in a separate chip, providing greater design flexibility but at the cost of lower bandwidths compared to PIM.

This chapter also delves into the details of PIM implementations based on different memory technologies, including DRAM, SRAM, and RRAM. It discusses various techniques and approaches for performing computation within these memory technologies, such as bit-line computation, LUT queries, and analog dot-product computations (using memristors).

It was highlighted that, while PIM is capable of attaining a higher bandwidth to memory, enabling massive parallelism, it is also highly constrained by the internal architecture of memory devices, limiting the design of NDAccs and their applicability, as well as requiring the programmer to know the low-level structure of the memory. On the other hand, PNM allows for much more design freedom, as the NDAccs reside in a logic die outside the memory controller, while still attaining remarkable performance and energy benefits.

Several PNM architectures are also explored, highlighting technologies like the HMC and the HBM that enable the integration of logic layers with memory arrays within the same chip. Examples of commercial PNM devices, such as UPMEM, Aquabolt-XL, and AXDIMM, are presented, showcasing their performance and energy efficiency benefits.

Additionally, multi-level NDP architectures are analyzed. These NDP-enabled architectures

can offload workloads to multiple levels of the memory hierarchy, depending on the locality of data and the potential benefits offered at each level. Some proposed architectures, like the ones by Ahn et al. [43] and Lockerman et al. [93], are discussed, highlighting their ability to dynamically determine the most suitable level for computation.

In addition, the existing development technologies for validating and assessing the performance of NDP architectures are surveyed. Namely, classic RTL development technologies, such as PiMulator [94], and simulation-based approaches using architectural simulators like gem5, ZSim, and Ramulator are presented. The advantages and limitations of these tools are discussed alongside recent proposals aimed at extending them to natively support NDP exploration.

Finally, the scarcity of comprehensive simulation-based NDP development technologies is emphasized, as well as the limitations of existing previous proposals, which are often restricted to specific NDP architectures or lack support for realistic scenarios involving the integration and simultaneous simulation of NDAccs with standard processing systems and memory hierarchies.

3

A Processing Near Memory Architecture for Signal Processing Applications

Contents

Driven by the negative impact of the widely-known memory wall issue in the signal processing applications (and particularly in memory-bound Machine Learning (ML) applications), the work presented in this chapter resources to Near-Data Processing (NDP) solutions to mitigate its effect, resulting in significant performance improvements. The proposed approach consists of a Cache Compute System (CCS) based on the solutions presented in [12, 13] that couples a Single Instruction Multiple Data (SIMD) unit to the Last-Level Cache (LLC) of a general-purpose ARM Central Processing Unit (CPU). The resulting system was modeled and evaluated using the widely-accepted gem5 architectural simulator. Differently from previous proposals [3, 6, 15, 31], this CCS architecture is based on robust fully-digital Functional Units (FUs), capable of implementing complex arithmetic, shift, and logic operations atomically. Furthermore, by being coupled to the LLC, it takes advantage of data locality and allows to solve important coherence issues that arise from doing computation directly in the main memory (updating data in the main memory may conflict with data present in cache). The presented research also includes a convenient library to program and control the CCS from plain C code.

3.1 System Overview

From the system point of view, the proposed CCS consists of a device that is tightly connected to the memory hierarchy of a conventional processing system. As shown in Figure 3.1, the CCS is connected to the processing system at two levels: (1) at the processor level, through which the processor communicates with the CCS by reading and writing the internal registers of the CCS' Programming Interface (PI); and (2) at the LLC level, using the memory bus of the shared LLC, from where the CCS issues data requests and outputs the commands' results. The following subsections detail the internal architecture of the proposed solution as well as how it operates.

3.1.1 Near-Cache Compute Structure

The proposed CCS is composed of several independent SIMD FUs that are managed by a common control unit, which allows operating over data directly fetched from the cache. This CCS is meant to be integrated with general-purpose (possibly multi-core) processors and their respective memory subsystems. Therefore, it includes all the necessary mechanisms to support the complexity of such systems: (1) from the processor side, the CCS receives commands from (any of) the processing core(s), indicating the addresses of the input and output data, the size of the operands, and the operation to be performed; (2) given the existence of a virtual memory subsystem, the CCS translates the virtual addresses that were provided by the CPU using its

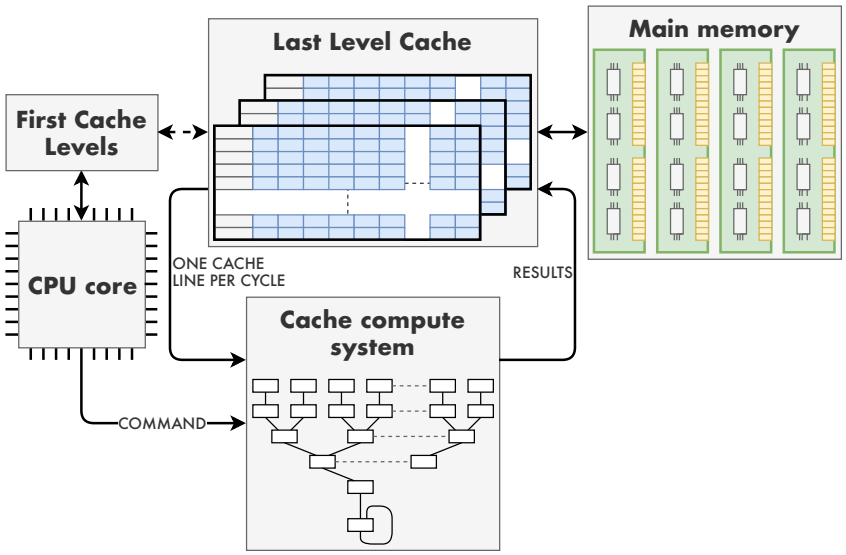


Figure 3.1: Block diagram of the proposed system integrating a processing core with the CCS.

own independent Translation Lookaside Buffer (TLB), which is synchronized with the processor's TLB; (3) the commands are executed by the order of arrival, and the CCS offers synchronization mechanisms to ensure that two sequential commands do not conflict; (4) the CCS gathers the data required by the executing command by issuing read requests to the cache, receiving an entire cache line atomically; (5) after executing the command, the CCS writes the results directly into the cache, one line at a time.

Hence, after receiving a command for execution, the CCS sends a data read request to the cache. If the requested data is present in the cache, it is retrieved immediately. Otherwise, the request is forwarded to the main memory. Naturally, while it waits for the requested data, the CCS enters an idle state, which (depending on the implementation) can be either executing no instruction or clock gating, disabling the CCS while it is not in use. For operations requiring two vector operands, the CCS fetches the operands sequentially. After gathering the first operand, the CCS stores it in a buffer and waits for the second operand. As soon as all the required data is available, the CCS starts executing the command.

Depending on the operation type, the execution can take from one to several clock cycles to complete. However, since the CCS is fully pipelined, it becomes available for receiving the next command right after all the previous operands were fetched and the previous command started executing. Figure 3.2 depicts two execution examples of commands that require different numbers of clock cycles to complete (Rectified Linear Unit (ReLU) and dot-product), as well as a top-level schematics of the CCS tree-shaped internal architecture composed of several levels of FUs (which is discussed in detail in Section 3.2). The first command (a) performs an element-wise operation

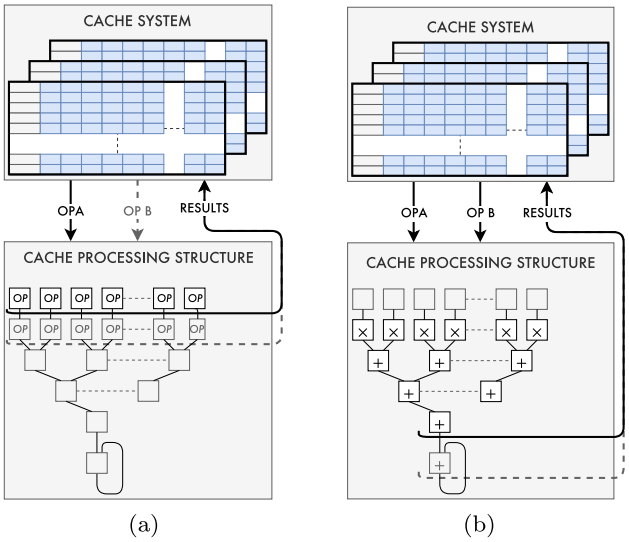


Figure 3.2: Execution of two different instructions in the CCS: (a) ReLU; (b) dot-product.

over all the elements of the vector operand, taking at most two clock cycles to complete. On the other hand, the second command (b) combines all the elements of the operands into a single one and takes a number of cycles at least equal to the number of levels of the tree-shaped CCS. When the execution terminates, the results are stored again in the LLC, invalidating any outdated data in the upper cache levels, and the processor is notified. If the operation executed by the CCS produces a vector as result (which is at most the size of a cache line), the elements are written back to the cache in parallel. For the currently supported instructions, the size of the output vector is the same as the operands'.

The proposed CCS essentially presents four advantages when compared to existing scalar or vector processors: (1) it makes extensive use of data locality, which is particularly advantageous in data-centric applications with regular memory access patterns; (2) since data is not moved all the way from the cache to the processing cores, less energy is spent on data transfers; (3) by exploiting the high bandwidth that is available near the cache, it is possible to achieve high-performance levels resulting from vectorization, by operating in up to an entire cache line in parallel; (4) due to the proposed CCS structure, FUs at different levels can be programmed to perform different operations, making it possible to execute complex commands.

3.1.2 Concurrent Operation

From the point of view of the host CPU, the CCS behaves like an independent co-processor that fetches, operates, and writes data in parallel without run-time intervention of the CPU. The only interactions between the CCS and the CPU happen when programming the CCS and in-

quiring its state (to determine whether the execution has finished and the results are available). Naturally, while the CCS is operating, the processor is free to execute a different workload, as long as there are no dependencies with the data being computed by the CCS. From a programming perspective, these dependencies are left to the programmer to resolve. Before reading or writing data that conflicts with the results calculated by the CCS, the programmer has to ensure that those results are available for being used. Otherwise, write-after-read and write-after-write conflicts may take place. Luckily, standard synchronization mechanisms can be used to work around this issue, such as mutexes, semaphores, and barriers.

Since the CCS operates at cache level, all the inherent coherence mechanisms are used. For instance, when writing the results to the cache, the existing coherence mechanisms will take care of automatically invalidating any outdated data in upper cache levels. Therefore, as long as the programmer guarantees correct synchronization between the commands running in the CCS and the workload being executed in the CPU, no other data hazards are possible.

3.1.3 Integration with Hierarchical Caches

On multi-level cache systems, the CCS may be integrated at any level of the cache hierarchy or even at multiple levels. For instance, several instances of the CCS can be coupled to each level of cache, maximizing the attained parallelism. On the other hand, a single instance of the CCS can be used to minimize the hardware and power overhead. In this respect, integrating the CCS with the LLC seems to be the most interesting option since: (1) it provides the maximum proximity with the data that resides in the main memory (allowing to reduce the traffic across the upper memory hierarchy); (2) usually, the LLC is the largest level of cache, which positively affects the number of expected cache hits; (3) since the LLC is usually shared by the several cores, all the issues that could arise from writing to the cache are automatically solved by invalidating the corresponding addresses when writing to the LLC, which will ensure that the data will be fetched to upper-level caches when read again by the CPU cores. Moreover, operating at the LLC level may also reduce contention with the processing cores, which is useful for task-level parallelism.

In physical processing systems, the CCS is intended to be fabricated within the same chip as the LLC, i.e., in the CPU's System on Chip (SoC), as a chiplet, communicating with the LLC through the same mechanisms as the CPU (using conventional cache ports and interconnects), and being integrated with the CPU (for control purposes) as a memory-mapped device. Nevertheless, the CCS could also be integrated with the Dynamic Random-Access Memory (DRAM) using technologies such as the Hybrid Memory Cube (HMC) or the High-Bandwidth Memory (HBM), enabling Processing Near Memory (PNM) at the main memory level. However, integrating the

CCS with the main memory would also require to implement mechanisms to guarantee that the operands processed by the CCS are up-to-date, as well as to propagate the results computed by the CCS to the cache hierarchy. On the other hand, attaching the CCS to the LLC rather than the main memory automatically solves this issue, since the existing cache coherence mechanisms can be used.

3.2 The CCS Compute Architecture

The internal architecture of the proposed CCS is composed of several FUs, each implementing a set of arithmetic, shift, and logic operations. These FUs are micro-programmed by a central control unit, allowing to execute complex commands using multiple FUs in a SIMD manner.

3.2.1 Programming Interface

To instruct the CCS to execute a command, the processor programs the CCS by writing in the memory-mapped registers of its PI. Each command contains information about the address of the operands to be processed, the computation to be performed, and the destination (in memory) of the result. The definition of the commands as well as the the underlying hardware architecture to support them was done having into account the specific requirements of common signal processing workloads, such as Convolutional Neural Networks (CNNs) and clustering applications. The CCS supports 47 distinct commands (which are summarized in Table 3.1) that can be classified according to three different criteria:

1. The class of mathematical operations: integer arithmetic, shift, or logic;
2. The type of the operands: two vectors (VOP2), a vector and a constant (VCOP), or a single vector (VOP1);
3. The class of functional operation: *map* or *reduce*.

Table 3.1: CCS' currently supported commands classified regarding their class of mathematical operations, type of operands, and class of functional operation.

		Instr	Description	
Arithmetic	VOP2	ADDVV	$r[i] = a[i] + b[i]$	*map*
		SUBVV	$r[i] = a[i] - b[i]$	
		MULVV	$r[i] = a[i] \times b[i]$	
		SSDVV	$r = \sum_i (a[i] - b[i])^2$	*reduce*
		SADVV	$r = \sum_i \lvert a[i] - b[i] \rvert$	
		IPVV	$r = \sum_i a[i] \times b[i]$	
	VCOP	ADDVC	$r[i] = a[i] + k$	*map*
		SUBVC	$r[i] = a[i] - k$	
		MULVC	$r[i] = a[i] \times k$	
	VOP1	COMP2V	$r[i] = -a[i]$	
		ADDV	$r[i] = \sum_i a[i]$	*reduce*
		MAXV	$r[i] = \max(a)$	
		MINV	$r[i] = \min(a)$	
		LESSVC	$r[i] = \begin{cases} 1 & , a[i] < k \\ 0 & , otherwise \end{cases}$	*map*
		GRTRVC	$r[i] = \begin{cases} 1 & , a[i] > k \\ 0 & , otherwise \end{cases}$	
		EQUVC	$r[i] = \begin{cases} 1 & , a[i] = k \\ 0 & , otherwise \end{cases}$	
		SQV	$r[i] = a[i]^2$	
		ABSV	$r[i] = \lvert a[i] \rvert$	
		RELUV	$r[i] = \begin{cases} a[i] & , a[i] > 0 \\ 0 & , otherwise \end{cases}$	
Shift	VOP2	SLLVV	$r[i] = \text{sll}(a[i], b[i])$	
		SRLVV	$r[i] = \text{sll}(a[i], b[i])$	
		SLAVV	$r[i] = \text{sla}(a[i], b[i])$	
		SRAVV	$r[i] = \text{sra}(a[i], b[i])$	
		ROLVV	$r[i] = \text{rol}(a[i], b[i])$	
		RORVV	$r[i] = \text{ror}(a[i], b[i])$	
	VCOP	SLLVC	$r[i] = \text{sll}(a[i], k)$	
		SRLVC	$r[i] = \text{srl}(a[i], k)$	
		SLAVC	$r[i] = \text{sla}(a[i], k)$	
		SRAVC	$r[i] = \text{sra}(a[i], k)$	
		ROLVC	$r[i] = \text{rol}(a[i], k)$	
		RORVC	$r[i] = \text{ror}(a[i], k)$	
Logic	VOP2	ANDVV	$r[i] = a[i]$ and $b[i]$	
		NANDVV	$r[i] = a[i]$ nand $b[i]$	
		ORVV	$r[i] = a[i]$ or $b[i]$	
		NORVV	$r[i] = a[i]$ nor $b[i]$	
		XORVV	$r[i] = a[i]$ xor $b[i]$	
		XNORVV	$r[i] = a[i]$ xnor $b[i]$	
	VCOP	ANDVC	$r[i] = a[i]$ and k	
		NANDVC	$r[i] = a[i]$ nand k	
		ORVC	$r[i] = a[i]$ or k	
		NORVC	$r[i] = a[i]$ nor k	
		XORVC	$r[i] = a[i]$ xor k	
		XNORVC	$r[i] = a[i]$ xnor k	
	VCOP	NOTV	$r[i] = \text{not}\, a[i]$	
		ANDV	$r[i] = a[0]$ and $\cdots$ and $a[B-1]$	*reduce*
		ORV	$r[i] = a[0]$ or $\cdots$ or $a[B-1]$	
		XORV	$r[i] = a[0]$ xor $\cdots$ xor $a[B-1]$	

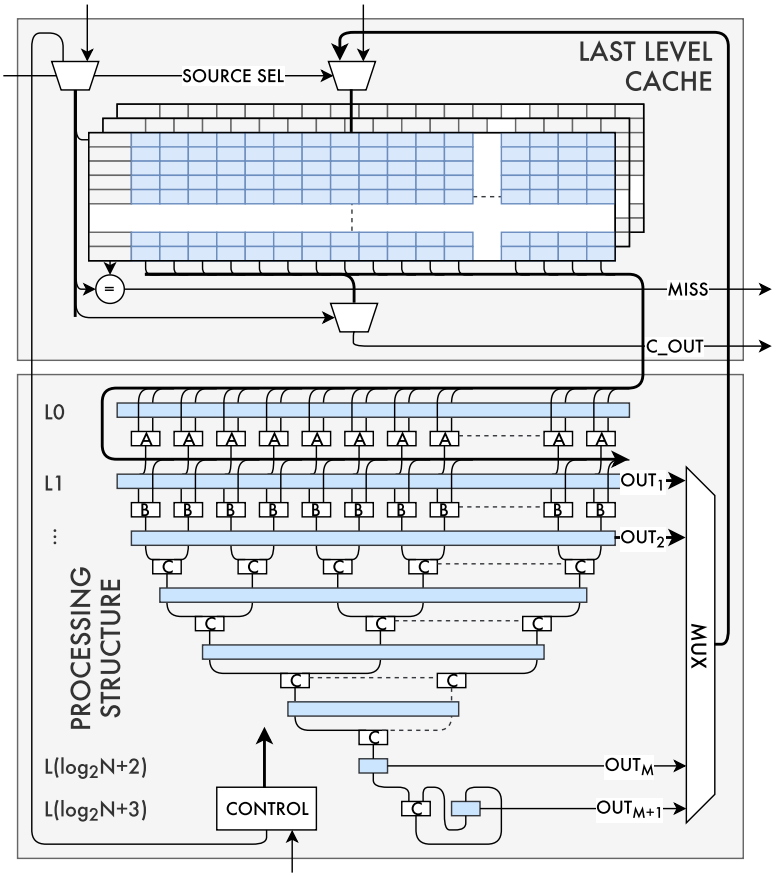


Figure 3.3: Processing structure of the CCS.

3.2.2 Compute Infrastructure

The CCS datapath is composed of several levels of quasi-generic FUs arranged in a pipelined binary tree, as shown in Figure 3.3. To ensure the aimed performance and energy gains, some architectural aspects have been considered concerning (1) the integration of the CCS with the conventional memory subsystem, and (2) the CCS' internal compute structure.

3.2.2.A Integration with the Data Cache

Differently from most general-purpose processors, the CCS does not read/write a single word from/to the cache. Instead, it fetches/stores an entire cache line atomically. For the VOP2 commands (see Table 3.1), the CCS sequentially fetches the two vector operands using the same input bus. First, one of the operands is fetched from the cache and stored in a buffer. Then, the other operand is fetched, fulfilling all the requirements for starting execution.

Whenever the operands are not in cache, they are automatically fetched from the main memory to the cache, similarly to what would happen if they were requested by a general-purpose

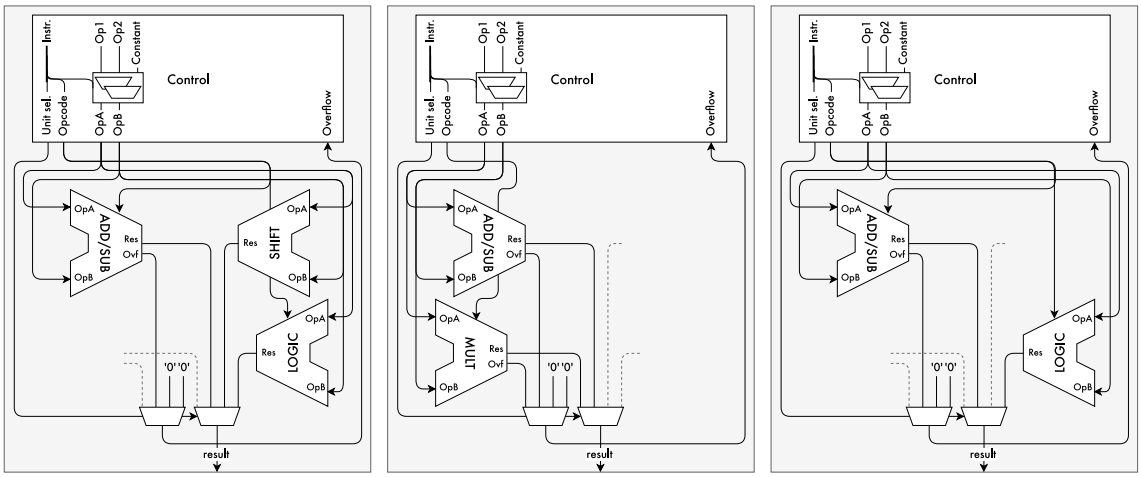


Figure 3.4: FUs that compose the proposed CCS: (a) first level units, integrating an adder/subtracter, a shifter and a logical unit, (b) second level units containing an adder/subtracter and a multiplier, and (c) remaining FUs, including only an adder/subtracter and a logical unit.

processor. Since most LLCs are not only larger but also have higher associativity than upper cache levels, the data being evicted due to loading the operands required by the CCS will most probably be the least used. Thus, loading the vector operands from the main memory does not necessarily increase the number of collisions between the data being used by the processing cores and those used by the CCS.

3.2.2.B Pipeline Functional Units

To maximize the CCS' throughput, the processing structure integrates a pipelined binary tree, with each level implementing different arithmetic, shift, and logic operations, depending on the command being executed.

The operands (fetched from the cache) are delivered to the first level FUs and the computation starts as soon as all the operands are available (see Figure 3.3).

The pipelined CCS datapath is composed of three different types of FUs, as shown in Figure 3.4. The first level is composed by type A FUs (see Figure 3.4a), integrating an adder/subtracter, a shifter and a logic unit. The first level of the CCS is responsible for several element-by-element operations, including: `ADD`, `SUB`, `COMP2`, `SLL`, `SRL`, `SRA`, `ROL`, `ROR`, `AND`, `NAND`, `OR`, `NOR`, `XOR`, `XNOR`, `NOT`, and element-wise comparison.

The second level of the CCS is composed by type B FUs (see Fig 3.4b), being capable of computing the following operations: `MUL`, `SQ`, and `ABS`. Type B units are the only that include an integer multiplier, since many commands supported by the proposed CCS require, at most, one multiplication. The multiplier was included in the second (rather than the first) level because

commands such as SSDVV and SADVV, typical in machine learning and multimedia applications, require one operation before multiplication. Other common operations that require multiplication (e.g. IPVV) do not require any operation before multiplication (thus, skipping the entire first level of the CCS), but require a reduction step afterward, which is compatible with the used approach.

All the other pipeline stages of the CCS are only used by commands of type *reduce*. These computing stages are equipped with type C FUs (see Figure 3.4c), which only implement arithmetic (except multiplication) and logic instructions: ADD, MAX, MIN, AND, OR, and XOR. For these units, only an adder/subtracter and a logical unit are required.

The operation at each level is controlled by control signals that are decoded by the CCS from the command issued by the processor.

3.2.2.C Pipeline Dataflow

There are four computing levels of the CCS that produce output. The command being executed determines which of the four outputs is valid and when. The first level produces the output for all operations of type *map* that do not involve multiplication (which is only implemented in the second level). Therefore, the output of *map* operations involving integer multiplication may only be collected in the second level. The output of operations of type *reduce* can be produced in two levels of the CCS: the last and the second to last. If the vector operands involved in the *reduce* operation fit within a line of cache (i.e., the operands are entirely processed in a single run of the CCS), then the output is collected in the second to last level. Otherwise, the operands cannot be processed in a single run of the CCS, since there are not enough resources to process all the elements simultaneously (the operands are too large). In that case, the command is automatically split into multiple pipelined operations, and the last level of the CCS is used to accumulate the sub-results of each operation until all elements are processed.

3.2.3 Programming the CCS

The programming of the CCS by the processor is done using the provided PI. This interface is composed of memory-mapped registers through which the CPU specifies the command to be executed, an optional constant, and the addresses of the operands and the result. Additionally, there is a memory-mapped register in the CCS' PI through which the processor instructs the CCS to start executing the command. As soon as all the operands required by the command are fetched to the CCS, the computation starts. When it finishes, the results are written back to the cache and the processor is informed that the results are available through another register in the CCS' PI.

Listing 3.1: Implementation of a 2D convolution using the provided CCS library.

```
#define L_SIZE          // size of cache line
#define DATA_WIDTH      // width of data matrix
#define DATA_HEIGHT     // height of data matrix
#define KERNEL_LENGTH   // size of the kernel

external int **data;    // data addr
external int *kernel;   // kernel addr
external int **result;  // result addr
int a[L_SIZE];          // data buffer

// 2-D convolution
for (int i = 0; i < DATA_WIDTH; i++) {
  for (int j = 0; j < DATA_HEIGHT; j++) {
    gather_data(a, i, j);
    ccs_setup(IPVV, KERNEL_LENGTH, 0, a, kernel,
        result + i * DATA_WIDTH + j, 1);
    ccs_start();
  }
}
ccs_wait();
```

To facilitate the process of writing the parameters into the programming registers and controlling the CCS, a fully optimized library is provided that exports a set of routines that can be straightforwardly used by the programmer. The library has the following four routines:

1. `ccs_setup(<args>)` writes the following parameters (`<args>`) of a command to the CCS' PI:
   - `cmd_id`: command identifier;
   - `op_len`: length of the vector operands;
   - `k`: optional constant;
   - `opa_addr`: start address of the first vector operand;
   - `opb_addr`: start address of the second vector operand;
   - `res_addr`: start address of the result;
   - `stride`: stride of the operands (and the result for applications of type *map*).
2. `ccs_start()` instructs the CCS to start executing the command previously configured in its PI;
3. `ccs_check()` verifies if the previous command has finished and the CCS is ready to receive a new command;
4. `ccs_wait()` hangs until the previous command has finished and the CCS becomes ready to receive a new command.

Listing 3.1 shows the code of a two-dimensional convolution using the library provided with the CCS. The method `gather_data` is responsible for gathering the elements of `data` corresponding

to the convolution window at a given moment.

All the developed routines were tuned at assembly level to ensure that the maximum optimization is achieved. For example, Listing 3.2 illustrates the implementation of these low-level routines using ARM assembly.

3.3 System Evaluation

To assess the performance benefits of the proposed CCS, a gem5-based simulation environment was developed. The following subsections explain the evaluation methodology and discuss the main experimental results.

3.3.1 Methodology

An accurate model of the proposed CCS was developed and integrated with the gem5 model of an in-order ARM CPU equipped with a NEON SIMD unit. The CPU model was fine-tuned by considering the parameters found in the gem5-X framework [18] to produce reliable results regarding the ARM Cortex-A53 CPU, and the CCS was connected to the CPU through its data port and to the LLC (in this case the L2 cache) using the existing memory bus. Since the architecture of the CCS is highly pipelined (and additional pipeline stages can still be added if necessary to further reduce the critical path without affecting performance), it was considered that the CCS can operate at the same frequency of the CPU core.

An additional module was also used to forward the memory requests from the CPU to the memory subsystem or between the CPU and the CCS' PI, depending on the target address. If the target address of the request is reserved to the CCS' PI, the module forwards the request to the CCS. Otherwise, the request is forwarded to the memory subsystem. The translation of the virtual addresses of the operands into their corresponding physical addresses (at the cache) is ensured by a dedicated TLB that is synchronized with the processor's TLB, as explained in Section 3.1.

The conceived CCS model was designed to target fixed-point vector operations. Since the size of a cache line in the reference CPU is 64 bytes wide, the developed model of the CCS can process up to 64 bytes of data per clock cycle. In this evaluation, the CCS was configured as a vector FU capable of processing up to 16 words of 32 bit each, 32 16-bit words or 64 8-bit words.

Listing 3.2: Routines written in ARM assembly to program and control the CCS from C code.

```
void ccs_setup (uint64_t cmd_id,
  uint64_t op_len,
  uint64_t k,
  uint64_t opa_addr,
  uint64_t opb_addr,
  uint64_t res_addr,
  uint64_t stride) {
  // store parameters in the programming registers
  __asm( "str %[cmd_id],   [%[addr_cmd_id]]\n\t"
         "str %[op_len],   [%[addr_op_len]]\n\t"
         "str %[k],        [%[addr_k]]\n\t"
         "str %[opa_addr], [%[addr_opa_addr]]\n\t"
         "str %[opb_addr], [%[addr_opb_addr]]\n\t"
         "str %[res_addr], [%[addr_res_addr]]\n\t"
         "str %[stride],   [%[addr_stride]]\n\t"
   : : [cmd_id]        "r" (cmd_id),
       [addr_cmd_id]   "r" (CCS_CMD_ID),
       [op_len]        "r" (op_len),
       [addr_op_len]   "r" (CCS_OP_LEN),
       [k]             "r" (k),
       [addr_k]        "r" (CCS_K),
       [opa_addr]      "r" (opa_addr),
       [addr_opa_addr] "r" (CCS_OPA_ADDR),
       [opb_addr]      "r" (opb_addr),
       [addr_opb_addr] "r" (CCS_OPB_ADDR),
       [res_addr]      "r" (res_addr),
       [addr_res_addr] "r" (CCS_RES_ADDR),
       [stride]        "r" (stride),
       [addr_stride]   "r" (CCS_STRIDE)
   );
}

void ccs_start () {
  const uint64_t start = 0x1;

  __asm( "str %[start], [%[addr_start]]\n\t"
   : : [start]      "r" (start),
       [addr_start] "r" (CCS_START)
   );
}

uint64_t ccs_check () {
  volatile uint64_t ready;

  __asm( "ldr %[ready], [%[readiness_addr]]\n\t"
    : : [ready]          "r" (ready),
        [readiness_addr] "r" (CCS_READINESS)
    );

  return ready;
}

void ccs_wait () {
  return;
  while (check() != 0x1);
}
```

3.3.2 Experimental Results and Discussion

To assess the performance of the proposed CCS, two different evaluation procedures were considered. First, a set of microbenchmarks was executed, each corresponding to a single CCS command involving vector operands of the size of a cache line, and the results were compared with those obtained using the CPU core. Second, six kernels commonly used by CNNs and clustering algorithms were evaluated. Five of the kernels are commonly used by CNNs. The sixth kernel consists of the distance computation phase of the k-Nearest Neighbors (kNN) clustering algorithm, which is also common to other algorithms (e.g., *k*-Means). The tested kernels as well as the used parameters are presented in Table 3.2.

The CPU baseline was fully optimized using all the compiler optimizations, including vectorization support. Additionally, it was considered that the operands were stored in the memory hierarchy level closer to where the computation took place (L1 cache when using the CPU and LLC when using the CCS).

Figure 3.5 shows the results obtained for each microbenchmark.

As expected, commands of type *map* take fewer cycles to complete than commands of type *reduce*, and, overall, the CCS shows significant performance benefits across all the microbenchmarks. The microbenchmark associated with the highest performance improvement was the `ROLVV` command, which requires several operations when executed by the processor's Arithmetic

Table 3.2: Parameters of the kernels used to assess the benefits offered by the CCS.

		Parameter	Value	Number of runs
Conv	**1D**	Data size *(length)*	1000 *(1000)*	1000
		Kernel size	15	
	2D	Data size *(length)*	$\{100, 100\}$ *(10000)*	10000
		Kernel size	$\{3, 3\}$	
	3D	Data size *(length)*	$\{10, 10, 10\}$ *(1000)*	1000
		Kernel size	$\{3, 3, 3\}$	
Max Pool		Data size *(length)*	$\{99, 99\}$ *(9801)*	1089
		Patch size	$\{3, 3\}$	
		Stride size	$\{3, 3\}$	
ReLU		Data size *(length)*	$\{100, 100\}$ *(10000)*	10000
kNN		*k*	4	1000
		Distance	`SSDVV`	
		Training	1000	
		Testing	1	
		Features	16	
		Classes	8	

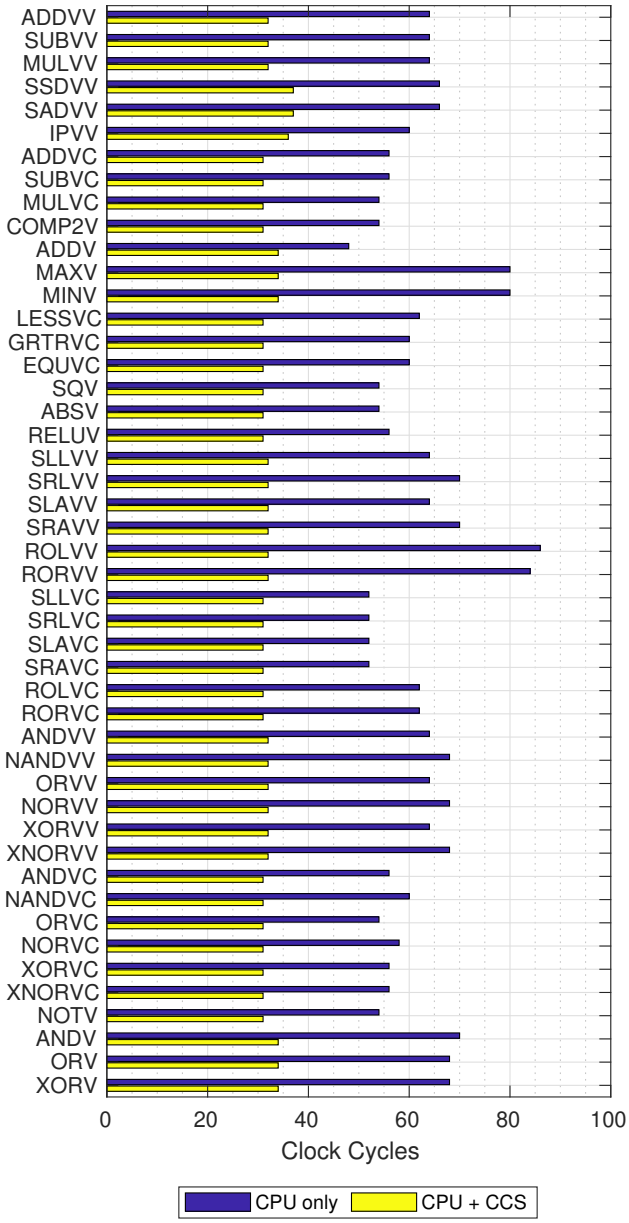


Figure 3.5: Performance results of a set of microbenchmarks, each one corresponding to a CCS command, when executed using only the reference CPU or the CCS.

and Logic Unit (ALU), but a single one when implemented by the CCS.

Also, Figure 3.6 shows the variation of the performance gains with the size of the operands for a command of type *map* (`ADDVV`) and a command of type reduce (`ADDV`). As expected, the performance gains increase with the size of the operands due to the pipeline architecture of the CCS. When the vector operands are too big, requiring a command to be split in several runs, the CCS issues the requests for the operands in sequence, one per cycle, and starts performing the computation as soon as the operands of the first run arrive. Since the requests for the operands of the second run were issued right after the requests for the operands of the first run, the second

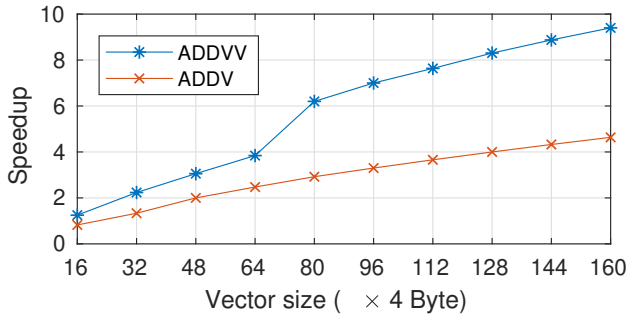


Figure 3.6: Variation of performance gains with the size of the operands for a command of type *map* (`ADDVV`) and a command of type reduce (`ADDV`).

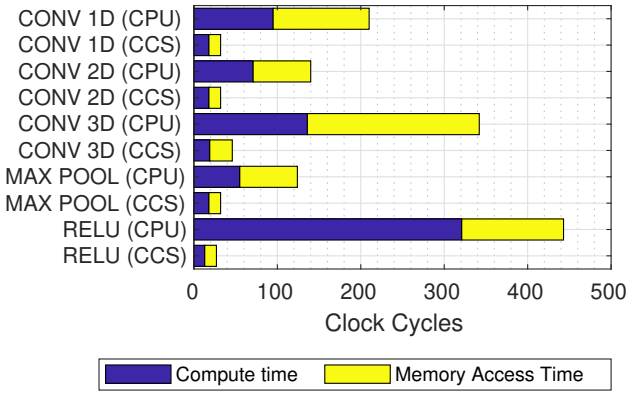


Figure 3.7: Performance results regarding five kernels commonly used by CNNs, showing the time used to access the memory and the actual computation time.

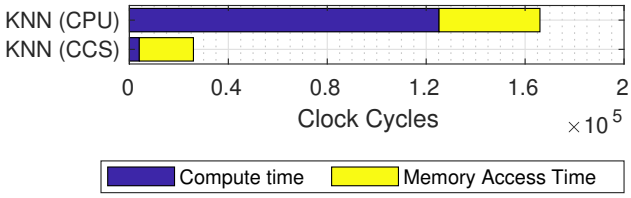


Figure 3.8: Performance results regarding the distance computation phase of the kNN algorithm, showing the time used to access the memory and the actual computation time.

run can be executed immediately after the first, in a pipelined manner. Therefore, the executing time of the two runs is much lower than twice the execution time of a single run. On the other hand, processing a vector operand twice as big in the CPU takes approximately twice as many cycles.

Figures 3.7 and 3.8 depict the execution time associated with the processing of the six kernels using vector operands with 32-bit elements. For visualization purposes, the results are normalized to the number of cycles required to process 64 bytes of data. As expected, the CCS allows for significant performance benefits on all tested kernels. In particular, the ReLU kernel presents the highest speedup of 40.6× (for vectors of 8-bit elements, as shown in Figure 3.9) due to its greater

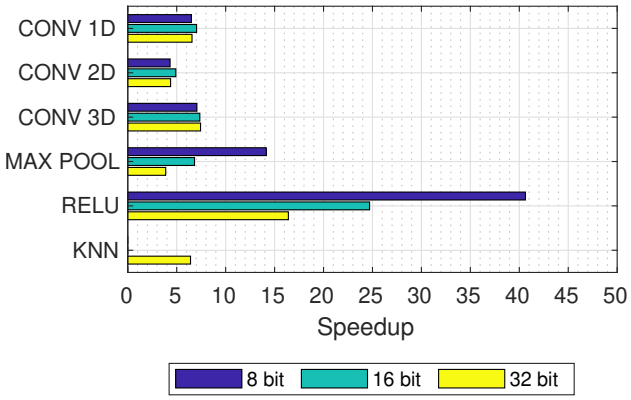


Figure 3.9: Obtained speedup by executing the six considered kernels using the CCS.

complexity when executed on a general-purpose CPU, contrasting with the single execution cycle required in the CCS. It is also worth mentioning that, although processing times are still dominated by memory accesses, the time spent by the CCS accessing memory is significantly lower than that of the CPU. Therefore, the performance benefits associated with the CCS have two complementary sources: (1) fewer memory accesses when compared to a conventional general-purpose CPU; and (2) more efficient implementation of complex operations that may otherwise take tens of cycles on a conventional general-purpose CPU. Furthermore, it is also noticeable that the convolution benchmarks executed by the CPU do not attain higher speedups when using vectors of smaller elements (8-bit and 16-bit) (see Figure 3.9). This is due to the fact that the NEON SIMD engine (used when executing kernels on the CPU) is capable of simultaneously operating four 32-bit elements, eight 16-bit elements or sixteen 8-bit elements. Therefore, both the CCS and the NEON engine consume the same amount of data per cycle regardless of the operands' representation. However, the Max Pool and the ReLU benchmarks do not use the SIMD unit of the processor due to their complex implementation. Therefore, reducing the size of the elements to half increases the number of clock cycles required to process a 64-byte vector. On the other hand, reducing the size of the elements to 16-bit for the Max Pool and the ReLU benchmarks increases the speedup attained by the CCS to 1.7× and 1.5×, respectively. Reducing the precision of the operands even further to 8-bit increases the performance speedup of the CCS to 3.7× and 2.5×, respectively.

Finally, to further evaluate the advantages brought by the novel CCS, the proposed experimental evaluation also considered an alternative accelerating infrastructure similar to those that are typically found in a general-purpose computer, namely Graphics Processing Units (GPUs). In accordance, parallelized versions of the previously discussed benchmarks were produced and executed in two different GPU devices (GPU 1: GeForce GTX 1080 Ti; GPU 2: TITAN RTX).

Table 3.3: Thousands of cycles spent by the six studied kernels when executed using only the CPU, the CCS, or a GPU device (GPU 1: GeForce GTX 1080 Ti; GPU 2: TITAN RTX)

		CPU only	CPU + CCS	CPU + GPU 1			CPU + GPU 2		
				Kernel	Transfers	Total	Kernel	Transfers	Total
Conv	1D	210	32	236	2124	2360	42	504	546
	2D	1406	320	108	2952	3060	20	408	428
	3D	343	46	278	1882	2160	38	330	368
Max Pool		135	35	94	1744	1838	16	304	320
ReLU		4440	270	26	946	972	12	308	320
kNN		166000	25891	62000	194000	456000	56000	96000	152000

Table 3.3 shows the performance results (in thousands of clock cycles) obtained while executing the kernels using these GPUs and compares them with the CPU and the proposed CCS.

In general, it was observed that the GPUs perform significantly worse than the CPU-only and the CCS systems, which can be explained by the fact that the tested kernels are memory-bound. Hence, while the CCS focus on processing at the DRAM's peak bandwidth with a low latency, the GPU's performance is highly limited by the additional overheads of transferring data between the external DRAM and the GPUs' global memory. Therefore, even though the GPU's relative performance improves for larger datasets, it is always bound by data transfers for the CCS' application use cases. Furthermore, processing on the GPUs leads to an overhead associated with the launching of threads in the GPU.

On the other hand, the CCS allows a much higher bandwidth to the memory, spending virtually no time transferring the operands and storing the results, and has a negligible programming overhead.

Furthermore, the available hardware in GPU devices highly surpasses that of the CCS. Naturally, this imposes a significant overhead in terms of both the global system's hardware and energy requirements, which is not always compatible with the system's constraints (e.g., edge devices which are highly constrained in terms of both hardware and energy supply).

3.4 Summary

In this chapter, a novel architecture to explore the performance benefits of NDP is proposed. Contrasting with previous proposals, the proposed CCS does not rely on the internal hardware of memory devices to perform computation. Instead, it makes use of fully digital FUs to accelerate signal processing applications (although it can also be used for general-purpose computing). The devised CCS is meant to operate allied with a common general-purpose CPU, and operates over data directly fetched from the cache, leveraging both proximity to the data (fetching and storing

data becomes less time-consuming) and the simultaneous access to an entire cache line.

For evaluation purposes, the CCS was integrated with a high-performance ARM Cortex-A53 CPU and simulated using the gem5 architectural simulator. A comprehensive software framework was developed to program and synchronize the CCS from plain C code. Results show that the CCS provides significant performance benefits in the form of reduced memory communications and increased processing power. Six kernels commonly used in the context of CNNs and clustering algorithms were tested, with the obtained speedups ranging from 3.9× to 40.6×. Additionally, GPU-parallelized versions of the tested kernels were used to compare the performance of the CCS with conventional GPU devices, which allowed to conclude that the CCS presents a better alternative than a GPU whenever the application is memory-bound.

4

A Full System Solution for the Development of Near-Data Processing Architectures

Contents

In the previous chapter, the architecture of the proposed Cache Compute System (CCS) was modeled, validated, and evaluated using the widely-accepted general-purpose architectural simulator gem5 [21]. However, this process highlighted an urgent problem regarding the Near-Data Processing (NDP) design exploration: the lack of standard tools and procedures to support the early development and evaluation of these devices.

The performance evaluation of Near-Data Accelerators (NDAccs) is critically different from that of standard accelerators since the behavior of the memory devices and the cooperation with the host Central Processing Unit (CPU) highly affects their performance. Moreover, directly prototyping such accelerators by developing Register Transfer Level (RTL) code is generally complex, expensive, and time-consuming, in particular when the intended goal is to simply evaluate the potential of an idea, or to adjust the parameters of an architecture still under consideration. Additionally, without properly integrating the accelerator with the remaining system, such methodologies may lead to inaccurate results due to control and synchronization overheads or the contention generated at the memory level attributed to the co-existence of several processing devices simultaneously accessing the memory hierarchy.

During the past decade, several contributions have been made to tackle the challenges involved in the conception of pre-RTL development tools, targeting heterogeneous high-performance processing systems. They point out important problems, such as the lack of sufficiently accurate models and the scalability of existing solutions [4, 5, 67, 100–102]. Although relevant proposals on design exploration and performance prediction were presented, they usually target fairly homogeneous systems (such as conventional multi-core CPU systems [103–105]), and are hardly suited for heterogeneous systems featuring specialized hardware accelerators, such as NDAccs. Furthermore, they are also unsuited for early design exploration phases (where the parameterization of the architectures is not fixed yet), as a detailed specification of the device is usually required [17].

In addition, most artifacts used in the literature are either not optimized to evaluate NDAccs (e.g., in [15], the authors use a Graphics Processing Unit (GPU) simulator to evaluate their NDP architecture), are limited to few architectures [18], or depend on post-simulation steps to combine the performance metrics of the NDAccs with the remaining system (resulting in the inability to simulate the simultaneous operation of the CPU and the NDAcc) [16, 19]. This not only makes it difficult to establish comparisons between such evaluation methodologies but may also bring into question the validity of the obtained results.

In this chapter, an original NDP simulation toolchain providing full support to develop, validate,

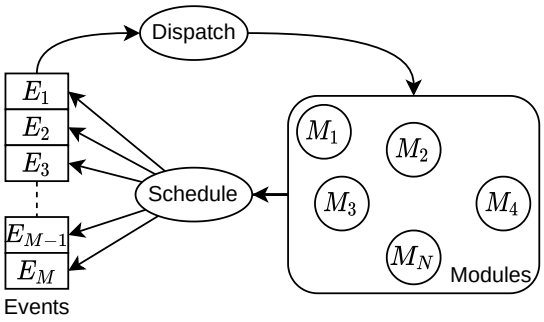


Figure 4.1: Event-driven simulation principle. M_i, where $i \in \{1, \ldots, N\}$, represent the models of the different hardware modules of a processing system in gem5. When a module receives input from another module, it schedules an event to be triggered after a number of clock cycles corresponding to its latency that awakens the module that consumes that output.

and evaluate the performance of new custom NDAcc architectures is discussed. This toolchain, henceforth designated NDPmulator, is based on the gem5 architectural simulator [21] and provides full support for simultaneous simulation of the CPU, memory subsystem, and NDAccs, simple integration of NDAccs with processing systems, and memory configuration mechanisms.

4.1 NDPmulator Overview

NDPmulator is based on the widely established gem5 architectural simulator [21], an event-driven simulator that delivers cycle-accurate results and capable of emulating the latency associated to each block in the processing system, as well as the inherent data contention. Accordingly, the latency associated with data movements or hardware operations is simulated through modules triggering events that involve other modules after the number of cycles corresponding to the latency of the simulated circuit, which will then schedule more events, as illustrated in Figure 4.1. While an extensive list of modules can be simulated out of the box in gem5, simulating custom hardware architectures requires the implementation of the corresponding models. This task is particularly complex for NDAccs due to the need to couple them to an existing CPU and memory hierarchy, which requires to implement the packet communication protocol needed to exchange data with the CPU (for control purposes) and the memory hierarchy (for obtaining operands and storing results).

In order to facilitate this process, NDPmulator offers three main components. First, an architectural framework implementing a programming interface and a data load/store unit is provided to easily connect the custom NDAccs with the existing CPU and memory hierarchy. A custom NDAcc based on this provided model has access to functions that allow to communicate with these structures, making the complex protocols required for implementing control and data transfers trans-

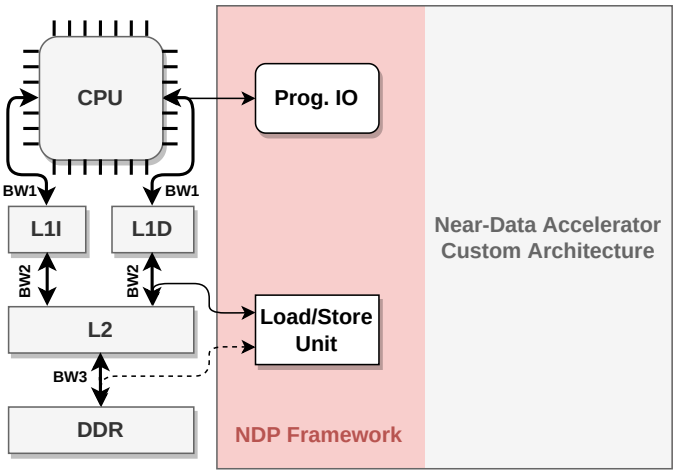


Figure 4.2: Physical representation of the proposed NDP architectural framework. BW1, BW2, and BW3 represent configurable NDPmulator parameters that define the bandwidths between the different levels of the memory hierarchy.

parent to the developer, while still allowing fine-grained control over data movements. Second, convenient gem5 simulation scripts are also provided to automatically connect NDAccs to the CPU, map their programming interfaces into the CPU address space, and connect them to one or more levels of the memory hierarchy. NDPmulator includes scripts for simulating bare-metal processing systems featuring NDAccs–System Emulation (SE) mode–, or executing a standard Operating System (OS)–Full System (FS) mode. Finally, NDPmulator includes a simple and organized programming paradigm for developing applications using NDAccs based on the provided architectural framework. NDPmulator also includes Linux device driver templates to be used with FS mode based on [106].

4.2 NDP Architectural Framework

The architectural framework of NDPmulator provides two main mechanisms to simplify the integration of custom NDAccs with standard processing systems, as depicted in Figure 4.2. The first mechanism is a Programming Interface (PI), which implements programming registers to provide a direct interface between the CPU and the NDAcc for control purposes. The size and total amount of registers are configurable parameters of NDPmulator, and it is up to the developer to define the purpose of each register, as these parameters depend on the implemented architecture. The second mechanism is a Load/Store Unit (LSU) that retrieves the operands from memory and stores the results by breaking down the high-level memory requests performed by the NDAcc into simpler requests that are sent to memory. By doing so, the LSU conceals the complex communication protocols used to transfer data between the memory devices and the

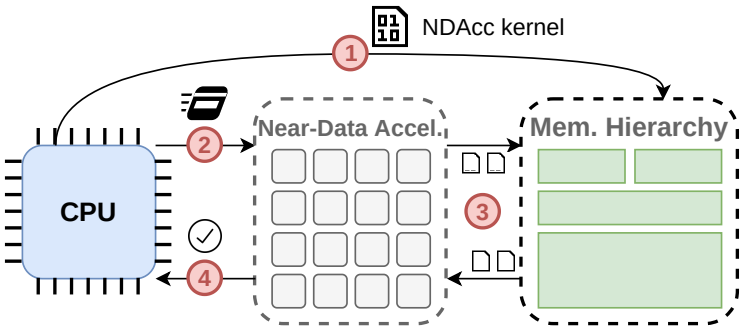


Figure 4.3: Operation of an NDAcc and interaction with the surrounding system: (1) the CPU stores the NDP kernel in memory; (2) the CPU triggers the NDAcc to start operating; (3) the NDAcc exchanges operands and results with the memory hierarchy; (4) the CPU polls the NDAcc.

processing elements, automatically dealing with failed request attempts (e.g., due to contention) and granting the delivery of the requested data to the NDAcc/memory.

It is important to mention that NDPmulator's architectural framework does not restrict in any way the internal architecture of the NDAccs. In particular, it is unaware of their Intruction Set Architecture (ISA) (and corresponding implementation) and the internal control required for their proper function (e.g., scheduling hardware). Moreover, although it provides the hardware infrastructure required to implement the communication between the CPU and the NDAcc, it is up to the developer to establish the communication protocol between them. The user may choose to implement a fine-grained control over the NDAcc (e.g., issue atomic instructions by reading or writing to the PI), or a more decentralized control where the NDAcc acts similarly to a co-processor (e.g., order the NDAcc to execute an entire pre-compiled kernel fetched from memory), as illustrated in Figure 4.3. In this example, the CPU starts by storing the NDP kernel (i.e., the sequence of instructions that will be decoded and executed by the NDAcc circuitry) into memory (step 1). Then, it writes the memory address where the NDP kernel was stored into the NDAcc programming registers (step 2) and triggers the NDAcc to fetch the kernel from memory and start executing it (step 3). While the kernel is being executed, the NDP input data is fetched from the memory hierarchy, and the results are stored back. During this time, the CPU is free to execute a different workload, since both devices can simultaneously operate on different data. Finally, the CPU polls the NDAcc programming register, waiting for the kernel completion (step 4).

Furthermore, NDPmulator allows to connect an NDAcc to any level of the memory hierarchy or even to multiple levels, and it includes mechanisms to configure the bandwidths of each connection (see Figure 4.2). The simulation of systems featuring multiple NDAccs is also supported.

In addition, by being based on gem5, NDPmulator is modular and can be extended to support specific evaluation requirements of custom NDAcc architectures. For example, specific metrics

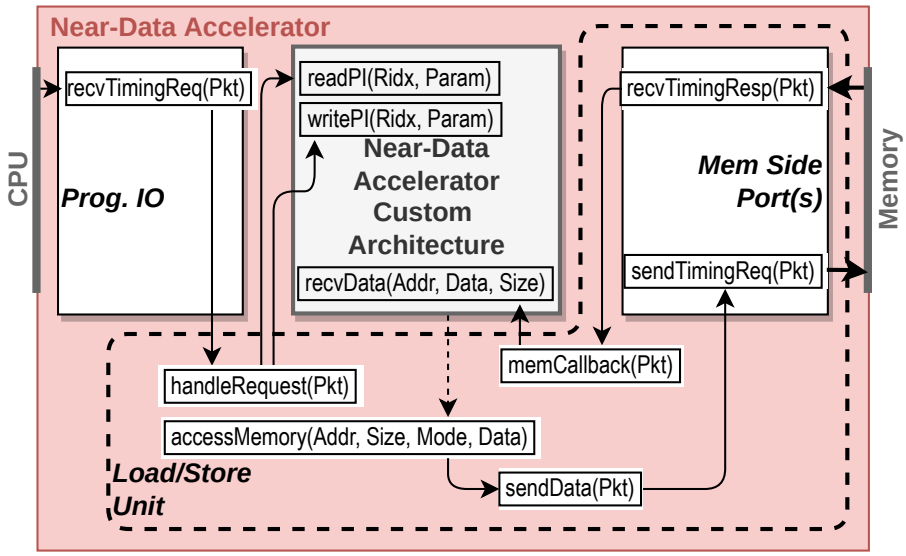


Figure 4.4: Function diagram of NDPmulator showing the relation between the framework software routines and the corresponding physical components illustrated in Figure 4.2.

can be obtained regarding the operation of certain modules within the custom NDAcc architecture. Also, specific hardware and power models can be created for posterior processing using McPAT [52] and/or CACTI [96].

4.2.1 CPU-NDAcc Programming Interface

One of the features provided by the NDPmulator's architectural framework is the generation of a PI, which is responsible for implementing a direct communication channel between the CPU and the NDAcc for control purposes. The number and size of the registers within the PI are configurable parameters of NDPmulator, and their purpose is defined by the developer, according to the requirements of the implemented custom NDAcc architecture. The PI is connected directly to the CPU through the data cache port, acting as a slave of the processing core, and its registers are mapped into the CPU address space. As illustrated in Figure 4.4, whenever the CPU accesses the PI registers for reading or writing, the corresponding request is automatically processed by the offered framework module, and the methods `readPI` and `writePI` are executed to read or write the accessed programming register.

It is worth emphasizing that the PI module offered by NDPmulator architectural framework does not impose any restrictions on the internal architecture of the NDAccs. Specifically, it is unaware of their control flow and corresponding implementation, as well as the implemented operations and ISA. It is up to the developer to specify the programming paradigm of the NDAcc, its architecture, and programming paradigm (e.g., using existing programming languages for het-

erogeneous systems such as SYCL [107]). Furthermore, while actual data can be exchanged between the CPU and the NDAcc through the PI, it is only effective for small (scalar) operands and/or results. For retrieving large amounts of data to be processed and storing the corresponding results, NDPmulator provides a high-performance communication lane that connects the NDAcc with the underlying memory hierarchy.

4.2.2 Load/Store Unit

The LSU features a memory port that enables the direct connection between the NDAcc and the memory hierarchy, for direct data access. Furthermore, it implements mechanisms that allow for the effortless retrieval of operands and storage of results produced by the NDAcc. Since memory models in gem5 have the same limitations as their corresponding physical devices, which impose a maximum request size, if a request for a burst of data to or from a memory device exceeds this maximum size, it must be broken down into smaller requests. To address this, NDPmulator's LSU automatically partitions large memory requests into smaller ones, each with a size that is pursuant to the constraints of the memory device (e.g., cache line size). These requests are then sent sequentially and at a pace that can be configured. Furthermore, since memory devices may be busy and reject requests, the LSU implements a First In, First Out (FIFO) request queue that retries sending the dropped requests when the memory device becomes available again. For example, if the NDAcc model needs to obtain an entire vector from memory, it can do so by simply providing the base address and the number of bytes required via the `accessMemory` method. Subsequently, the LSU schedules the necessary memory requests, arranges the data retrieved from memory in a buffer (as the memory requests may not be fulfilled in order), and returns it to the NDAcc model, as quickly as possible. In the case of a store operation, the LSU performs the necessary memory requests to store the entire vector in memory and informs the NDAcc model upon completion.

Figure 4.5 illustrates the operation of NDPmulator's LSU. First, the LSU verifies if the NDAcc request can be fulfilled in a single memory request. If that is the case, it creates a memory request of that size. Otherwise, it creates multiple requests with a size equal to or smaller than the maximum allowed size. Then, an attempt is made to send a single request to the memory device. If it succeeds and more memory requests are required to fulfill the original request made by the NDAcc, another attempt to send another request is scheduled. Otherwise, if it fails, the LSU waits until the memory device is ready again to retry sending the dropped request. This process is repeated until the original NDAcc request is fulfilled. When the memory returns a request response, the LSU matches the response with a previously issued request and stores the

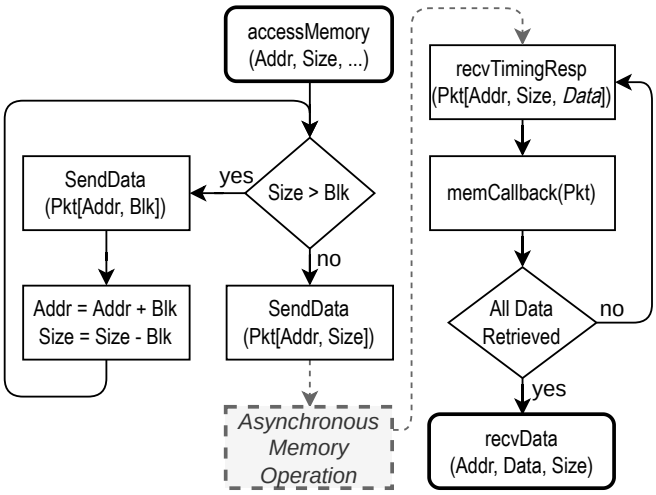


Figure 4.5: Diagram showing how NDPmulator's LSU unfolds large requests into smaller ones suitable to be sent to the memory devices.

retrieved data in the corresponding slot on a Re-Order Buffer (ROB). When the received memory response concludes a request made by the NDAcc, the corresponding data stored in the ROB is transferred to the NDAcc, and the memory transaction finishes.

Another important feature of the LSU offered by NDPmulator is that it can be coupled to any level of the memory hierarchy (or even multiple levels), with the corresponding bandwidths allowed at the different levels being configurable parameters. In addition, since when NDAccs are coupled with a cache and communicate through the same ports used by the CPU cores, all the usual cache coherence protocols are still supported. Thus, NDAccs modeled using NDPmulator are independent of the used cache coherence protocol. NDPmulator also offers the option of bypassing the memory controller, which lets developers adjust the latency and behavior of the memory devices. This feature is particularly interesting for Processing in Memory (PIM) design exploration, as it allows developers to emulate memory accesses using a custom latency. Additionally, NDPmulator is also compatible with having multiple NDAccs coupled to the same or different CPUs and memory devices.

An important aspect that has to be taken into account occurs when virtual memory mechanisms are considered since the data pointers made visible to user applications (virtual addresses) usually differ from those used to communicate with the memory hierarchy (physical addresses). Thus, in order to make this compatible with NDP, there needs to be an interfacing mechanism guaranteeing that, while the CPU application operates over virtual addresses, the NDAcc has access to the corresponding physical addresses to fetch the operands and store the results at the physical level.

4.2.3 Virtual Memory Translation and Management

In order to allow NDAccs to directly access the memory hierarchy for retrieving data and storing results, it is crucial that the memory addresses corresponding to the location where data is physically stored in the memory devices (physical addresses) are known to the NDAccs. However, from an application point of view, the physical addresses of data are usually not known. Instead, the application accesses data through a different type of address called virtual addresses (referring to the system's virtual memory), which usually differ from their corresponding physical addresses.

Virtual memory is an abstraction that provides an idealized view of memory to applications. It allows each process to have its own address space, starting from address 0, making it appear as though the process has the entire memory to itself. This abstraction simplifies programming and improves security and isolation between processes.

However, this abstraction only exists on the application side. At the memory subsystem level, only physical addresses are known. Therefore, the CPU is responsible for interfacing the virtual and the physical memory domains such that applications operate in their own virtual memory space using virtual addresses and memory requests at a physical level only use physical addresses. Figure 4.6 illustrates some of the different levels of abstraction of a processing system, from the lowest-level hardware to the highest-level software (user applications), highlighting the frontier between the levels that operate on virtual memory addresses and those that use physical memory addresses. While user applications above the OS' kernel level can only operate using virtual addresses, the underlying physical hardware infrastructure uses only physical addresses. At the OS' kernel level, both domains can be accessed, and it is at that level that the necessary translation between virtual and physical addresses occurs (supported on physical hardware structures provided by the CPU). To implement address translation, the CPU uses three key components:

1. **Page Table:** A data structure used to map virtual addresses to physical addresses. Each process has its own page table.
2. **Pages and Frames:** The virtual address space is divided into fixed-size blocks called pages. The physical memory is divided into blocks of the same size called frames.
3. **Memory Management Unit (MMU):** A hardware component responsible for handling the translation of virtual addresses to physical addresses. The MMU uses the page table to perform this translation.

Accordingly, the following steps are implemented by the CPU to perform address translation:

1. **Virtual Address Breakdown:** A virtual address is typically divided into two parts: (1) page

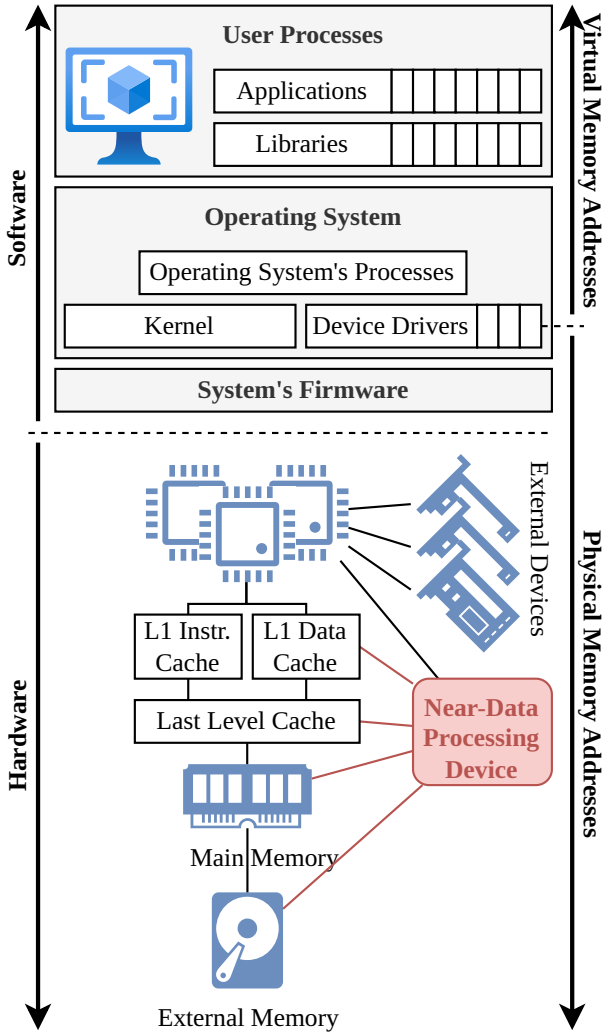


Figure 4.6: Different levels of abstraction of a processing system and the frontier between components that use virtual addresses managed by the CPU's virtual memory system and components that operate using physical addresses.

number, which identifies which page of the virtual address space the address is on; and (2) page offset, specifying the exact location within that page.

2. **Page Table Lookup:** The MMU uses the page number to index the page table, or locate it in the Translation Lookaside Buffer (TLB). If the required page is not present in the TLB, the MMU tries to fetch it from the page table stored in the main memory. The page table entry contains the frame number where the corresponding page is stored in physical memory. If the MMU is not able to retrieve the required page successfully from the main memory, a page fault occurs. This can happen for several reasons, such as when the page has been swapped out to the external memory to free up the main memory for other processes or when the program attempts to access an address that has not yet been allocated.

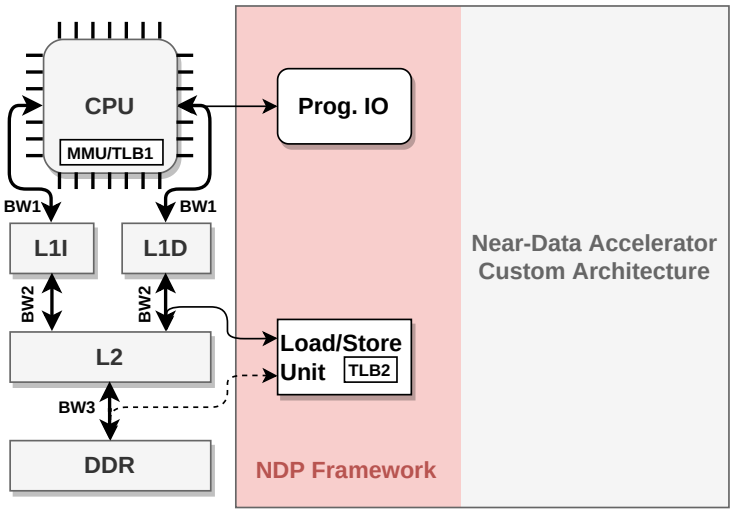


Figure 4.7: Physical representation of the proposed NDP architectural framework with a secondary dedicated TLB at the level of the NDAcc for explicit address translation.

3. **Physical Address Calculation:** The frame number from the page table entry is combined with the page offset to form the complete physical address. This physical address points to the exact location in the main memory.

Since NDAccs have to be able to fetch data from memory and store the results based on addresses communicated by the host application (i.e., virtual addresses), they also need to be capable of translating virtual addresses into physical addresses. To achieve this, NDPmulator uses two alternative approaches: explicit address translation and direct address mapping.

4.2.3.A Explicit Address Translation

Explicitly translating virtual addresses into physical addresses requires replicating the same hardware mechanisms implemented by the CPU for that purpose at the NDAcc level. In particular, NDPmulator's architectural framework implements a dedicated secondary TLB that shares the CPU's MMU, as shown in Figure 4.7.

Accordingly, the architectural framework of NDPmulator also includes methods to automatically use these translation mechanisms (transparently to the developer), as illustrated in Figure 4.8. Whenever the custom NDAcc issues a memory request, the framework asynchronously translates the address using the dedicated secondary TLB (`translateTiming`) and sends the request to memory with the corresponding physical address (`sendData`). Furthermore, NDPmulator reduces the number of accesses to the TLB by translating subsequent addresses within the same page by adding an offset to a physical address translated previously. Only when a page boundary is crossed, the address is translated by explicitly accessing the TLB again, as demonstrated in

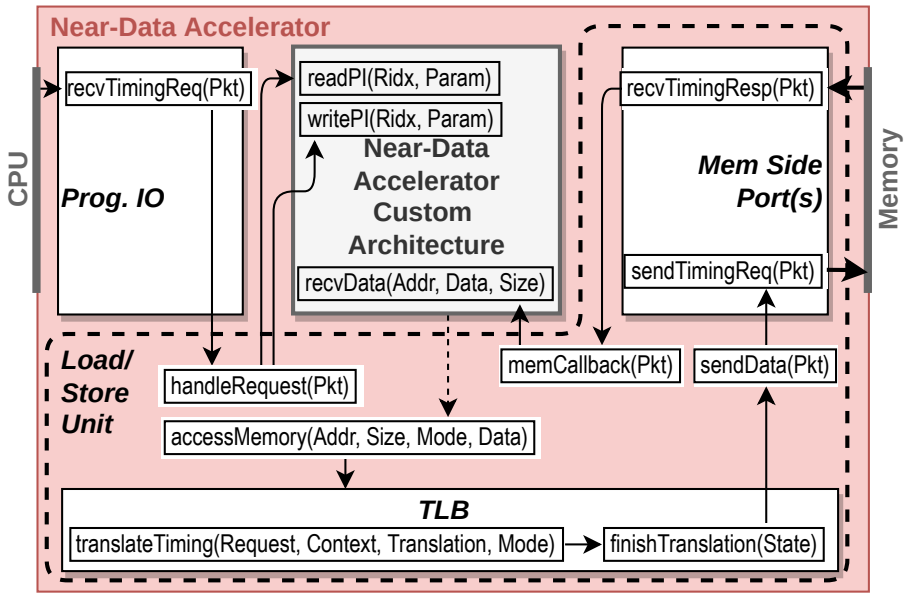


Figure 4.8: Function diagram of NDPmulator including methods to implement explicit address translation by using a dedicated secondary TLB at the NDAcc level.

Figure 4.9. In the event of a miss in the secondary TLB, the CPU's MMU is accessed to fetch the required page.

However, while versions of gem5 prior to 22.x (partially) supported the instantiation of the required hardware structures to enable explicit address translation by custom hardware accelerators (specifically in systems with an ARM core as the host CPU), this support was mostly deprecated in more recent versions of the simulator, making this unfeasible. Therefore, an alternative approach was employed which, by using a deterministic translation strategy, does not require dedicated address translation mechanisms.

4.2.3.B Direct address mapping

While reproducing the traditional explicit address translation mechanisms used by the CPU at the NDAcc level is arguably the most commonly used solution to cope with the existence of virtual memory [5], there is also the option of avoiding explicit address translation by employing a deterministic translation strategy, which does not require additional circuitry. In addition, this approach may also bring performance benefits, as the complex process of traditional address translation is replaced by a deterministic process with fixed (low) latency.

Hence, such a scheme is emulated by selecting a contiguous region within the existing physical memory and explicitly mapping it into an equally contiguous region in the virtual addressing space, such that the virtual and physical addresses in that region are equal. As a consequence, the translation of addresses from the virtual to the physical domains becomes unnecessary, and

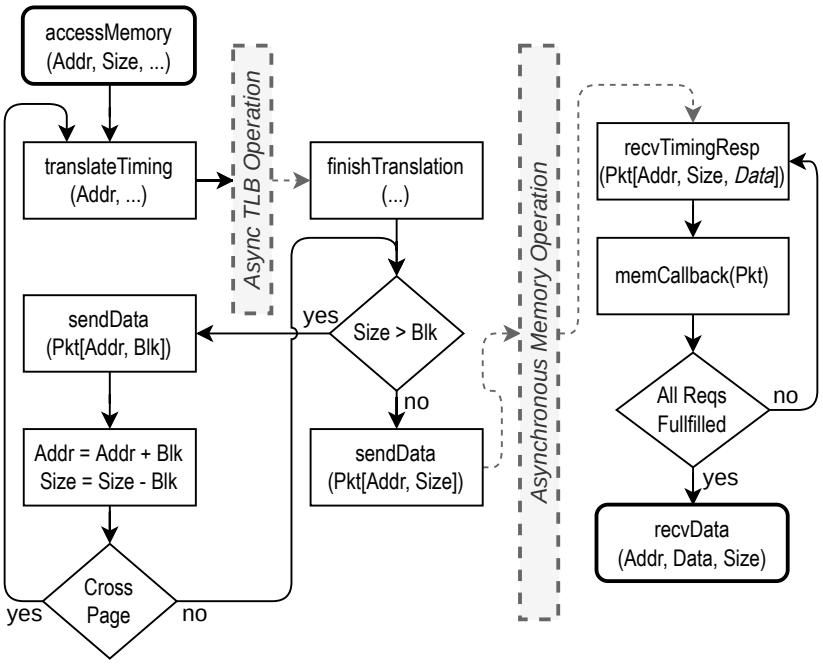


Figure 4.9: Diagram showing how NDPmulator's LSU unfolds large requests into smaller ones, translates the virtual addresses into the corresponding physical addresses, and sends them to the memory devices.

the NDAcc is capable of accessing data within this shared memory region without explicit address translation. In accordance, Figure 4.10 illustrates the corresponding NDPmulator's architectural framework and the system address management scheme. Region $[A, B]$ corresponds to the system's memory, where explicit translation mechanisms are enforced. Therefore, this region cannot be used for NDP purposes. Region $[B, C]$ corresponds to the address range where the NDAcc's PI is mapped, and region $[C, D]$ is directly mapped into the CPU's virtual memory, allowing to share this memory space with the NDAcc without replicating explicit address translation mechanisms at the NDAcc level.

It should be noted that such mechanisms do not relax any security aspects, which are still enforced to guarantee process isolation, preventing other processes from accessing CPU-NDAcc shared memory. In addition, the shared memory space between the CPU and the NDAcc has all the same features as the remaining addressing space, including being accessible by the CPU with the same latency as regular memory (since it exists in the same physical hardware). Hence, whenever required, this memory region can be used for purposes other than NDP, not being specialized for that single purpose (although NDPmulator would allow for that).

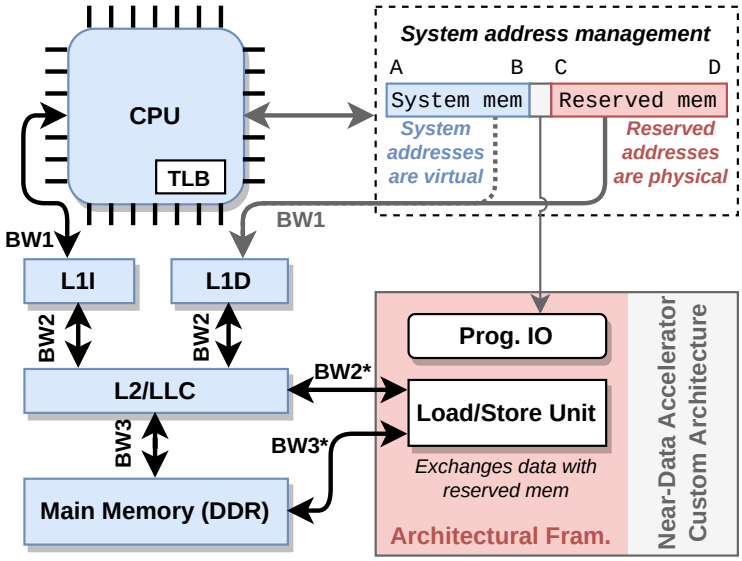


Figure 4.10: Physical representation of the proposed NDP architectural framework where the memory region shared between the CPU and the NDAcc is explicitly mapped into the CPU's virtual memory space, not requiring to implement mechanisms for explicit address translation at the NDAcc level.

4.3 Simulating NDAccs with NDPmulator

NDPmulator supports simulating NDAccs based on the previously described architectural framework using two different modes: System Emulation (SE) and Full System (FS). In SE mode, a single application using the NDAcc is simulated alone, without OS. Consequently, no OS-specific mechanisms and/or restrictions are considered. Naturally, this simulation mode tends to be optimistic, since all OS-related overheads that would otherwise exist are not considered, or reduced to the minimum. Also, there is virtually no contention on the use of resources (besides the bandwidth limit), which leads to a scenario where the NDAcc has nearly-exclusive access to the memory hierarchy. Nevertheless, since process isolation mechanisms are light in SE mode, it is still possible to read and write the PI registers of the NDAcc by simply accessing the physical addresses where they are mapped into the CPU addressing space.

In contrast, FS mode simulates the execution under the premises of a conventional OS (Linux), leading to results more similar to those that would be obtained in a real multi-tasking system, where the OS plays a crucial part in managing hardware resources and provides important security enhancements such as process and memory isolation. As a consequence, non-privileged applications cannot directly access the system's physical resources (due to security reasons). Thus, in order to establish communication between the user application and the NDAcc, the access to the NDAcc PI registers has to be delegated to the OS kernel through an appropriate device

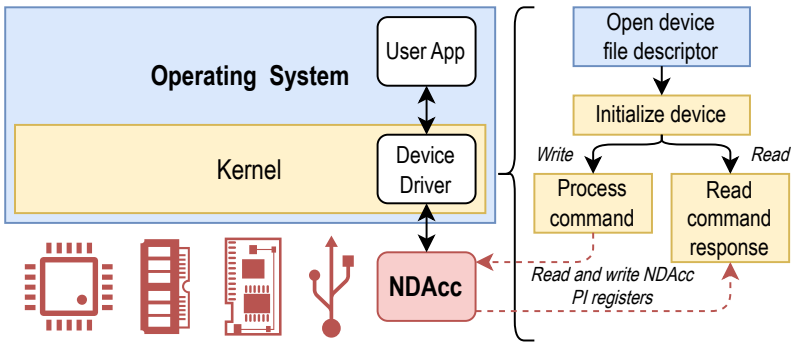


Figure 4.11: Diagram of a conventional NDP-enabled processing system featuring an OS. The device driver interfaces an application with the physical NDAcc by accessing its PI registers at the privileged kernel level.

driver (see Figure 4.11). However, this requires a more complex control flow on the application side, thus leading to extra programming overhead.

The following subsections describe the parameterization and simulation procedures corresponding to these two simulation modes (SE and FS).

4.3.1 SE Mode Simulation

NDPmulator provides two important mechanisms to simulate NDAccs using SE mode: (1) SE simulation scripts that connect the several system components; and (2) a programming paradigm/libraries that implement the routines required to control the NDAcc by reading and writing in its PI registers.

As illustrated in Listing 4.1, NDPmulator SE simulation scripts are logically divided into four parts. Part 1 describes the targeted processing system, in this case, featuring an Out-of-Order (OoO) CPU with two levels of cache and a DDR memory. Part 2 instantiates the included NDAcc using the NDPmulator's architectural framework, and connects it to the CPU and the L2 cache memory using crossbars (Part 3), whose bandwidth can also be adjusted to reflect the proximity of this device to memory through parameters such as the bus width of the interconnects (BW1, BW2, and BW3 in Figure 4.2) and the configuration parameters of the memory devices' models. Although in this example the NDAcc is coupled to the L2 cache, it can be as easily integrated with the L1 cache or the DDR memory. Full scripts exemplifying how to achieve this are available at `https://github.com/hpc-ulisboa/NDPmulator`. It is worth noting that this feature also enables the simulation of PIM accelerators, by configuring the allowed bandwidth according to the one commonly found inside memory chips. Furthermore, NDPmulator also provides detailed information regarding the memory transactions between the NDAccs and the memory hierarchy, which

Listing 4.1: Partial NDPmulator SE simulation script integrating a standard processing system and memory hierarchy with an NDAcc featuring explicit address translation mechanisms. In this example, the NDAcc is connected to the L2 cache. Nevertheless, NDPmulator allows to connect NDAccs to any memory device or even multiple devices.

```
# Standard processing system
system.cpu = DerivO3CPU()
system.cpu.icache = L1Cache()                                          (Part 1)
system.cpu.dcache = L1Cache()
system.l2cache = L2Cache()
system.mem_ctrl = DDR3_1600_8x8()

# NDAcc
system.ndacc = NDAcc(                                                  (Part 2)
    ndacc_rnge=('0x40000000', '0x40001000'))

class CustomL2XBar(L2XBar):
    def __init__(self):
        super(CustomL2XBar, self).__init__()
        width = 32  # 256-bit bus width (BW2)

# Connect NDAcc to CPU
system.ndacc.cpu_port = system.cpu.dcache_port                         (Part 3)
system.cpu.dcache.cpu_side = system.ndacc.mem_side

# Connect NDAcc to L2
system.l2bus = CustomL2XBar()
system.ndacc.dma_port = system.l2b.slave
system.l2cache.cpu_side = system.l2bus.master

# Map NDAcc PI into the host address space
system.cpu.workload[0].map(
    0x40000000,  # Host address space
    0x40000000,  # NDP device address space
    0x1000)      # Address range
                                                                       (Part 4)
# Emulated driver
system.ndacc_driver = NDAccDriver(
    hardware = system.ndacc,
    filename = "ndacc") # file descriptor

# Start simulation
exit_event = m5.simulate()
```

allows to determine the effective bandwidth between the two.

The provided scripts also exemplify how to map the NDAccs programming interface (provided by NDPmulator's architectural framework) into the CPU address space and configure an emulated driver when explicit address translation is used (also provided by NDPmulator's architectural framework) to implement the system calls required by virtual memory translation mechanisms (Part 4). However, when using direct address mapping, there is no need to implement an emulated device driver. Instead, the shared memory region between the CPU and the NDAcc is explicitly mapped into the CPU's virtual memory similarly to the PI, as shown in Listing 4.2.

Listing 4.2: Partial NDPmulator SE simulation script demonstrating how the CPU-NDAcc shared memory region can be mapped into the CPU virtual memory.

```
# Map NDAcc PI and CPU-NDAcc shared memory into the host address space
# MEMORY MAP:
# CPU-only region:          0x0         - 0x3fffffff
# CPU-NDAcc PI:             0x40000000 - 0x40000fff
# CPU-NDAcc shared memory: 0x40001000 - 0x7fffffff                    (Part 4)
system.cpu.workload[0].map(
    0x40000000,  # Host address space
    0x40000000,  # NDP device address space
    0x1000)      # Address range

# Start simulation
exit_event = m5.simulate()
```

To control the NDAccs from the program being executed in the CPU, NDPmulator also includes a straightforward programming paradigm, which allows to communicate with the NDAccs by reading and writing from/to their programming interfaces (provided by NDPmulator's architectural framework).

A simple C code example using an NDAcc whose operation flow is compatible with the one illustrated in Figure 4.3 targeting the SE mode is illustrated in Listing 4.3. Considering the version of NDPmulator that uses explicit address translation and replicates the translation mechanisms of the CPU at the NDAcc level, first, the NDAcc device driver is initialized, and the data pointer representing the NDAcc PI registers is set statically to the corresponding (physical) addresses (since there is no OS). Then, both the operands and the kernel are loaded into memory and the CPU instructs the NDAcc to start executing the kernel. It is important to mention that with explicit address translation being enabled, the operands can be stored anywhere in memory since the NDAcc is able to translate the virtual addresses passed by the CPU into the corresponding physical addresses. During the operation of the NDAcc, the CPU is free to execute other tasks. Finally, the CPU reaches the synchronization point where the results produced by the NDAcc are required for post-processing. In contrast, when using the version of NDPmulator that uses direct address mapping, there is no need for an emulated device driver, and the NDP operands (and results of operations) have to reside within the CPU-NDAcc shared memory region, where the NDAcc can determine physical addresses from virtual addresses directly without using explicit translation.

Listing 4.3: C code example to initialize and control an NDAcc using NDPmulator in SE simulation mode. The lines highlighted in red are exclusive to the version of NDPmulator that uses explicit address translation, while the green lines are specific to the version that uses direct address mapping. The routines `initData` and `initKernel` preload the data and the kernel into the memory region shared with the NDAcc. `__ndaccLaunchKernel` instructs the NDAcc to execute the kernel and `__ndaccReady` checks (pooling) if the NDAcc finished processing. Both these routines are implemented by the user who also defines the programming interface of the NDAcc.

```
int main() {
    // NDAcc driver initialization
    int fd = open("/dev/ndacc", 0);

    // Assign NDAcc memory addresses
    volatile void *ndacc_control = NDACC_CRTL_ADDR;
    volatile DATA_TYPE *dataset = (DATA_TYPE *) malloc(...);
    volatile void *kernel = (void *) malloc(...);
    ndacc_control[...] = dataset;
    ndacc_control[...] = kernel;
    volatile DATA_TYPE *dataset = NDACC_DATA_ADDR;
    volatile void *kernel = NDACC_KRNL_ADDR;

    // Dataset and kernel are prepared
    initData(dataset);
    initKernel(kernel);

    // CPU launches kernel on NDAcc
    __ndaccLaunchKernel(ndacc_control);

    // CPU processes tasks in background
    ...

    // CPU waits for NDAcc to finish
    while (!__ndaccReady(ndacc_control));

    // CPU post-processes the results
    ...

    return 0;
}
```

4.3.2 FS Mode Simulation

While SE mode is useful for validating and obtaining preliminary results of custom NDAcc architectures, it does not necessarily simulate the conditions of a realistic system due to the previously discussed limitations. On the other hand, FS mode provides a simulation scenario identical to a real system, allowing for a more thorough evaluation. Listing 4.4 illustrates relevant parts of an FS simulation script which describes a system identical to the one used in SE mode except that it also simulates a full-featured OS environment instead of simulating a single application in isolation. The complete FS simulation scripts can be found in the NDPmulator online repository (`https://github.com/hpc-ulisboa/NDPmulator`).

Listing 4.4: Partial NDPmulator FS simulation script integrating a standard processing system and memory hierarchy with an NDAcc.

```
ndacc = NDAcc(ndacc_rnge = (
        '0x40000000',   # Lower PI address                          (Part 1)
        '0x40001000'))  # Upper PI address

processor = SimpleProcessor(
    cpu_type=CPUTypes.TIMING, isa=ISA.X86,
    num_cores=1)

cache_hierarchy = NDAccCompatibleCacheHierarchy(
    l1d_size = "32kB", l1i_size = "32kB",
    l2_size = "256kB", ndacc = ndacc)                               (Part 2)

memory = SingleChannelDDR3_1600(size="2GB")

board = X86Board(
    clk_freq = "2GHz", processor = processor,
    cache_hierarchy = cache_hierarchy,
    memory = memory)

board.set_kernel_disk_workload(
    kernel = CustomResource("vmlinux"),
    kernel_args=[
        "earlyprintk=ttyS0", "console=ttyS0",
        "lpj=7999923", "root=/dev/sda1",                            (Part 3)
        # Reserve 1GB shared memory with NDAcc
        "mem=1G", "memmap=1G@0", "memmap=1G$1G"],
    disk_image =
        CustomDiskImageResource("root_fs.img"))

simulator = Simulator(board=board)
simulator.run()
```

Similarly to the previously referred SE simulation script (see Listing 4.1), Part 1 and Part 2 instantiate and connect the components of a system featuring an NDAcc, a CPU, a two-level cache hierarchy, and a Dynamic Random-Access Memory (DRAM). A crucial difference between SE and FS simulation scripts is that the latest is ISA specific, i.e., the components of the simulated system vary depending on the ISA. In this example, an x86 CPU is used. Nevertheless, NDPmulator also supports ARM and RISC-V ISAs for FS mode simulation.

Due to the presence of the OS security mechanisms, process isolation is enforced in FS mode. As a consequence, a user application cannot directly access the physical hardware and has to do so using the implemented device drivers, which act as bridges between the applications and the simulated hardware devices, translating commands issued by the user applications into instructions that the NDAccs can understand (and vice-versa). First, the user application requests access to an NDAcc by sending a request to the OS. Then, the OS assigns that device to the requesting application and prevents any other process of acquiring its lock. The device driver

Listing 4.5: C code example to initialize and control an NDAcc using NDPmulator FS mode. Similarly to the SE mode, the routines `initData` and `initKernel` preload the data and the kernel into the memory region shared with the NDAcc. `__ndaccLaunchKernel` instructs the NDAcc to execute the kernel and `__ndaccReady` checks if the NDAcc finished processing. Both these routines are implemented by the user who is also responsible for implementing a device driver suitable for the NDAcc custom programming interface, for which NDPmulator provides templates and documentation.

```
int main() {
    // Open file descriptor (device driver)
    int *fd_ndacc = open("/dev/ndacc", O_RDWR | O_SYNC);

    // Allocate memory in CPU/NDAcc shared region
    DATA_TYPE *dataset = ndaccAlloc(...);
    void *kernel = ndaccAlloc(...);

    // Dataset and kernel are prepared
    initData(dataset);
    initKernel(kernel);

    // CPU launches kernel on NDAcc
    __ndaccLaunchKernel(fd_ndacc, dataset, kernel);

    // CPU processes tasks in background
    ...

    // CPU waits for NDAcc to finish
    while (!__ndaccReady(fd_ndacc));

    // CPU post-processes the results
    ...

    return 0;
}
```

initializes the NDAcc and prepares it for communication. Finally, the application can send commands to the device driver, which will be translated (at kernel level) into low-level instructions that can be interpreted by the physical device (in this example, read and write accesses to the NDAcc PI registers). In addition, the device driver is responsible for handling errors that may occur during the communication between user applications and the physical devices, acting as an extra layer of security and preventing one or more components of the system to undergo a non-recoverable undefined state.

Listing 4.5 illustrates an example of a C application that uses an NDAcc in FS mode (equivalent to that of Listing 4.3). As it can be observed, this code already incorporates functionalities provided by the OS, such as the use of the file system. It should be noted that it is no longer allowed to statically set the pointers to the NDAcc PI registers nor the NDAcc shared memory region to the corresponding physical addresses due to process isolation mechanisms implemented by the

OS. Instead, the control of the NDAcc is implemented through a device driver (encapsulated by routines `__ndaccLaunchKernel` and `__ndaccReady`), and the memory regions to store the data to be processed and the kernel are also allocated (inside the CPU-NDAcc shared memory region) using OS mechanisms (wrapped by the method `ndaccAlloc`).

It is important to mention that while the NDAcc is processing data, it is up to the developer to enforce that no invalid data is accessed, i.e., the CPU does not try to access memory positions being read or written by the NDAcc. To ensure this, programs must be aware of what data is being used by the NDAccs, and implement proper synchronization using either high-level synchronization mechanisms (such as mutexes and barriers), or the value of status registers in the NDAccs programming interfaces. Furthermore, thanks to the existence of virtual memory and process isolation mechanisms, it is also enforced that processes other than the ones that have hold of the NDAccs cannot access the data being processed by them. All in all, NDPmulator offers guarantees that it is always possible to prevent the CPU to access memory regions involved in NDP computations.

In addition, NDP also requires maintaining data coherence across the several devices of the memory hierarchy. NDPmulator also deals with this important requirement of computing systems. When operating at the cache level, coherence is enforced by the existing cache coherence mechanisms (i.e., from the memory subsystem's point of view, the NDAccs are seen as processing cores), thus, it is transparent to the programmer. However, when working at the main memory level, the programmer has to manually flush the operands from the caches before the NDAcc can read them, and invalidate the memory region of the results in the cache before accessing them again from the CPU (can be done using the cache management instructions provided by the CPU). One can argue that manually evicting data from the cache deteriorates performance, which goes against the purpose of NDP. However, if the majority of the operands are stored in the main memory, evicting the few that are in the cache to process them near the main memory is preferred.

4.4 Case Studies

To demonstrate the viability of NDPmulator, the following subsections present and discuss two case studies: a simple near-data square root circuit, to exemplify the process of developing and integrating an NDAcc using NDPmulator; and a bidimensional NDP array using explicit address translation, providing preliminary evaluation results and comparing them with a classic evaluation approach.

4.4.1 Simple square root circuit

To illustrate the development process of an NDAcc and its integration with a standard processing system using NDPmulator, a simple near-memory square root module based on the circuit proposed in [108] is considered. This circuit receives as input a 32-bit floating-point number and calculates its square root in two steps: First, the exponent of the output is calculated by shifting the exponent of the input to the right by one bit; Second, the mantissa of the output is calculated by interpolating the coefficients stored in one of two local memories, depending whether the exponent of the input is even or odd.

To simulate this module using NDPmulator, the first step consists of creating a new C++ class (an architectural model) that inherits from the class representing the NDPmulator's architectural framework. As previously shown in Figure 4.4, NDPmulator requires the implementation of three main functions: `readPI` and `writePI`, which deal with read and write operations from/to the NDAcc's PI, and `recvData`, which deals with the response of requests to the memory. To send requests to the memory device, NDPmulator's architectural framework implements the routine `accessMemory`, which receives an address, the size of the data, an operation type (read or write), and the data to be written in case of a write operation.

Accordingly, the behavioral model of the simple near-data square root module is illustrated in Figure 4.12. First, the `writePI` method is called (multiple times), to transmit the following data to the NDAcc: base address of the operands, number of operands, base address to store the results. Then, the circuit will fetch batches of operands from memory, according to the availability of internal registers to store them, using the `accessMemory` method of the framework. The `recvData` method will be called as soon as the memory device returns a batch of operands to the NDAcc, which schedules them to be processed sequentially and stores the results in an output buffer. After all operands of a batch have been processed, the results are sent back to memory. Finally, an acknowledgment is received by the NDAcc indicating that the batch of results was successfully stored in memory.

Integrating the square root module with an existing processing system (at the L2 cache) is achieved with the steps illustrated in Listing 4.1 (for SE mode) or Listing 4.4 (for FS mode), while the host code that makes use of the NDAcc follows the structure described in Listing 4.3 (for SE mode) or Listing 4.5.

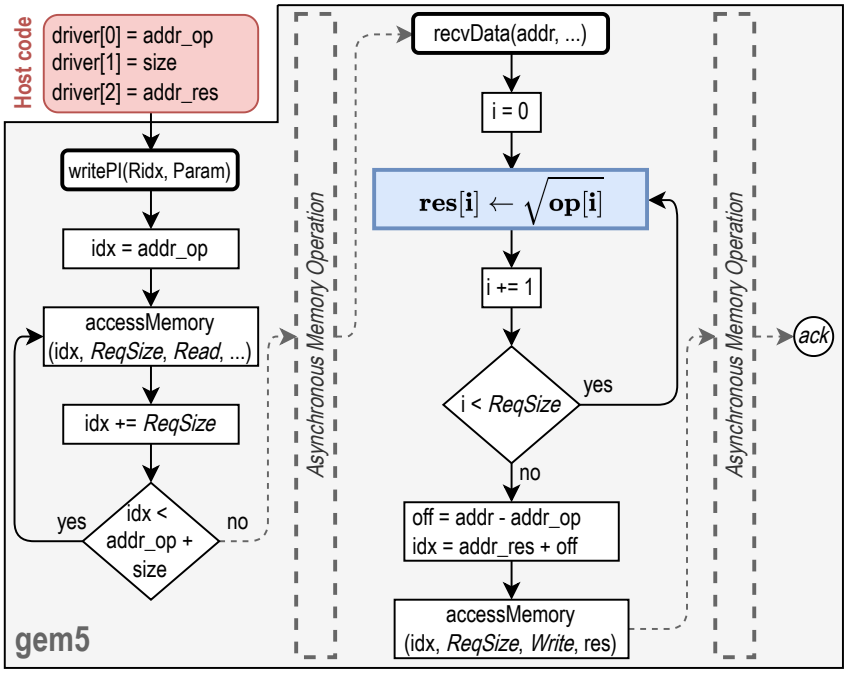


Figure 4.12: Behavioral model of the simple near-data square root module described in terms of calls to the methods defined by the devised NDPmulator's architectural framework.

4.4.2 Bidimensional NDP Array on LLC

In this second case study, a bidimensional processing array was implemented and connected to the memory hierarchy of a processing system, as shown in Figure 4.13. The considered NDAcc was integrated with the CPU through the memory-mapped interface and connected to the memory hierarchy through the mechanisms provided by the architectural framework of NDPmulator. Although the devised architecture was designed to be integrated with any level of the memory hierarchy, for this particular scenario, it was decided to couple it with the Last-Level Cache (LLC) to automatically maintain data coherence through the existing cache protocols.

The considered NDAcc features a dedicated LSU, responsible for fetching the input data and sending the results from/to the LLC through the data gather/scatter mechanisms provided by NDPmulator. The control module implements all the logic that allows the CPU to manage the NDAcc. It is also responsible for scheduling the NDP instructions to the PEs and keeping track of their status. The considered data path consists of a grid of similar PEs, capable of atomically processing a pair of vectors. PEs communicate with their neighbors through uni-directional buses: PE (i, j) communicates with PEs $(i+1, j)$ and $(i, j+1), i < N \wedge j < M$. Thus, the vector operands are loaded into the first row/column of PEs and propagated through the grid until reaching their

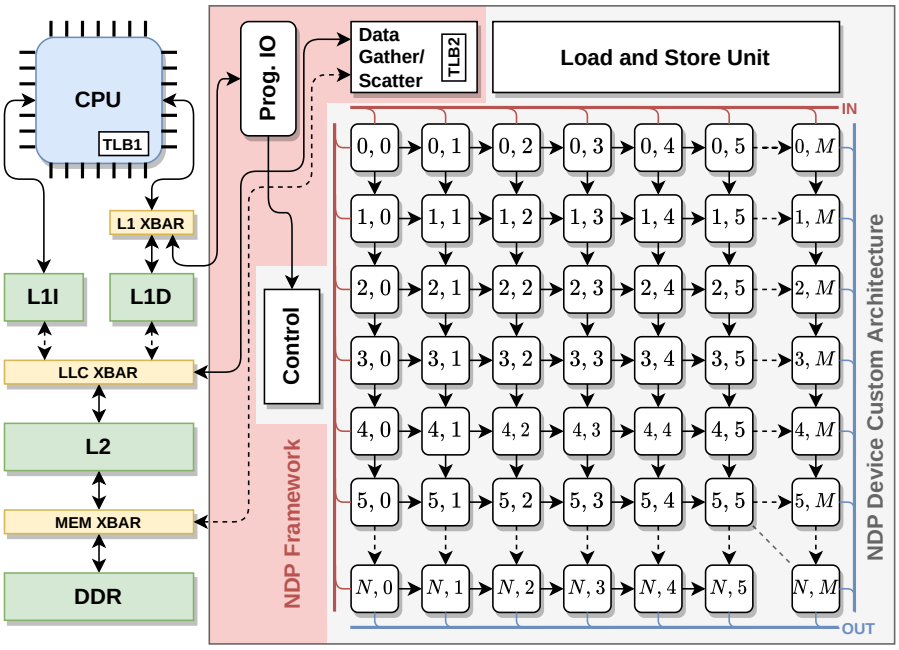


Figure 4.13: NDP architecture considered in the validation and evaluation of NDPmulator. The considered data path consists of a bidimensional array of similar Processing Elements (PEs).

destination. When available, the results are sent through the grid to the last row/column, and then collected by the LSU.

As shown in Figure 4.14, it is assumed that each PE is equipped with a 64-bit integer arithmetic unit, a 64-bit floating-point arithmetic unit (both implementing addition, subtraction, and multiplication), and a 64-bit logic unit. Additionally, the arithmetic units can be configured to process a pair of 64-bit operands, two pairs of 32-bit operands, four pairs of 16-bit operands, or eight pairs of 8-bit operands. Each PE also features four 64-bit internal registers, with three of them serving exclusively to store operands and an output register that can also store temporary results to be reused in subsequent operations. Since all PEs have the same internal configuration, a workload can theoretically be assigned to any cluster of PEs.

The considered host processing system was composed of a general-purpose CPU operating at $3.6\,\mathrm{GHz}$, whose model in gem5 accurately describes a high-performance ARM core. The caches were modeled after the ARM Cortex-A72 specification, featuring a $32\,\mathrm{kB}$ L1 data cache and a $1\,\mathrm{MB}$ L2 shared cache, connected through 256-bit-wide crossbars. The main memory consists of a DDR3 module operating at $1600\,\mathrm{MHz}$, with a 64-bit wide interface. For the majority of the experiments, the NDAcc was coupled to the shared L2 cache. However, for the last experiment (Section 4.4.2.C) it was also connected to the DDR memory.

To evaluate the advantages of the proposed simulation framework, three sets of experiments were performed. First, two arithmetic benchmarks (Matrix Addition and Matrix Multiplication) were

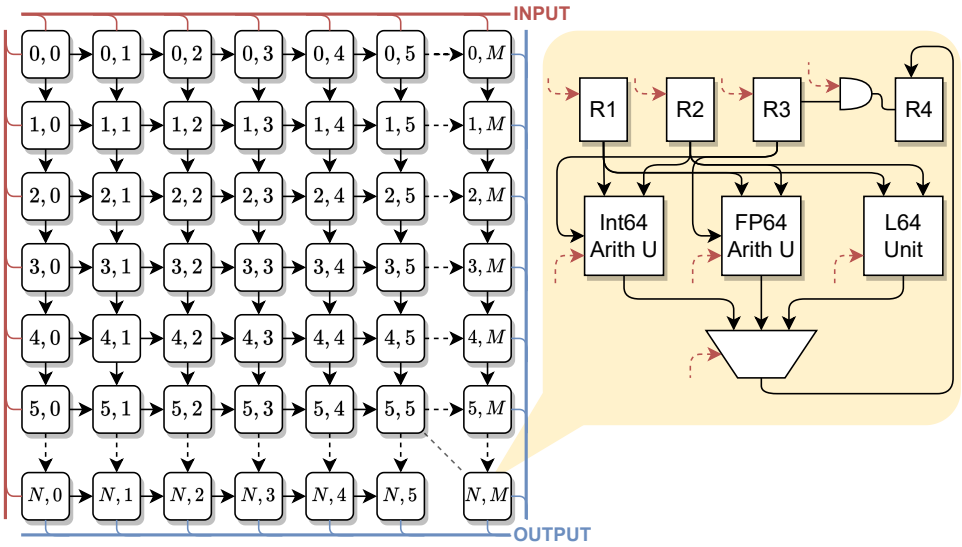


Figure 4.14: Considered internal architecture of each PE of the proposed NDAcc. Each PE includes Integer and Floating-Point Arithmetic Units and a Logic Unit.

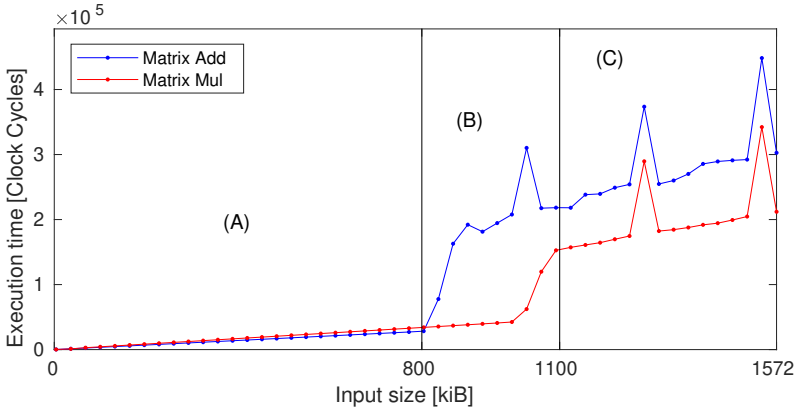


Figure 4.15: Performance of the NDAcc executing the Matrix Addition and Matrix Multiplication benchmarks for different matrix sizes (matrices that fit entirely in cache correspond to zone (A), and zones (B) and (C) involve matrices that do not fit entirely in cache).

executed by the NDAcc for multiple data sizes, allowing to observe its performance for different sets of data. Second, seven benchmarks from different application domains were executed by the processing system with and without the proposed NDAcc connected to the L2 cache, in order to evaluate the attained performance benefits. Finally, the obtained results using NDPmulator were compared with extrapolated performance values calculated using classical evaluation models.

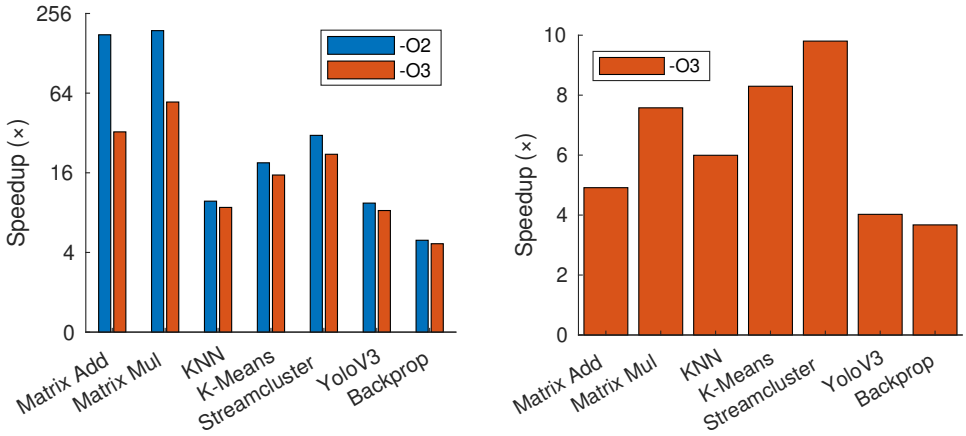
4.4.2.A Simple Arithmetic Benchmarks

The results of the first experiment are presented in Figure 4.15, showing the performance of the NDAcc while executing two regular arithmetic benchmarks: Matrix Addition and Matrix Multiplication. For this experiment each row of the matrices was 256 floating-point values long, and

the second matrix of the Matrix Multiplication benchmark was transposed in memory. As it can be observed, the resulting performance is not proportional to the size of the data being processed. In particular, for matrices larger than $1\,\mathrm{MiB}$ (which do not entirely fit in the L2 cache), the performance of the NDAcc depends on the DDR operation and cache eviction protocols (zones (A) and (B) of Fig 4.15). Furthermore, when executing the Matrix Addition benchmark simultaneously in the CPU and the NDAcc using $4\,\mathrm{KiB}$ matrices, a performance decrease of 2× was observed, with the average bandwidth between the NDP and the LLC decreasing from 173 bit/cycle to 84 bit/cycle. This effect is explained by the data contention caused by the CPU and the NDP, since both require large amounts of data from the memory subsystem. This shows that the performance benefits of the NDAcc can only be accurately predicted if the whole system is considered, including not only the NDAcc but also the memory hierarchy (which may not have a constant throughput), and the CPU (which manages the NDAcc and shares the same memory hierarchy, partially occupying its bandwidth). It is also noticeable that the Matrix Addition benchmark shows a significant performance degradation for smaller input matrices when compared with Matrix Multiplication. This is explained by the involved matrix sizes (not square), which impact the output matrices. In Matrix Addition, the output matrix size matches the input size filling up the cache sooner, as the L2 cache uses a write-back policy. In contrast, in Matrix Multiplication, the output matrix is smaller which allows using larger input matrices before triggering cache evictions that would otherwise penalize the performance. In zones (B) and (C) there is also the presence of local peaks of poor performance. These correspond to specific data sizes that, due to the cache eviction policy, generate more block evictions. Naturally, that leads to more requests being sent to the DDR, degrading performance.

4.4.2.B Mixed Benchmark Evaluation

The set of benchmarks used in the second experiment consist of seven applications from three different domains, as described in Table 4.1. For simplicity, the single-precision floating-point format is used across all the benchmarks, and all the vector operands involved in each accelerated kernel invocation are $4\,\mathrm{KiB}$ each, with two operands being required for each kernel. In the particular case of Matrix Addition and Matrix Multiplication, each row of the matrices is the same size as a cache line ($64\,\mathrm{B}$ or 16 floating-point values), and the second matrix of the Matrix Multiplication benchmark is transposed in memory. Furthermore, three baselines are considered for calculating the performance benefits introduced by the NDAcc: two using a high-performance in-order ARM CPU, with the benchmarks compiled with -O2 and -O3 optimization flags (Figure 4.16a); and one considering an OoO ARM core, with the benchmarks compiled with -O3 (Figure 4.16b).



(a) In-order CPU (b) OoO CPU

Figure 4.16: Performance benefits provided by the NDAcc for several benchmarks from different application domains.

Despite their differences, all seven benchmarks are highly parallelizable, with over $80\,\%$ of the workload being suitable to be executed in parallel. Due to its data-parallel compute capabilities and higher memory bandwidth (compared to the CPU), the NDAcc is able to significantly accelerate these regions, with overall performance speedups ranging from 3.7× up to 189.9×, as shown in Figure 4.16.

The performance improvements allowed by the NDAcc mainly depend on two factors: the fraction of the workload that can be accelerated by the NDAcc (the greater the accelerated workload, the higher the performance improvements–Amdahl's law); and the complexity of the kernel being executed by the NDAcc (kernels that reuse the outputs of the PEs in subsequent operations tend to allow higher speedups). For example, the parallelizable region of Streamcluster accounts for over $97\,\%$ of its total workload, and its corresponding NDP kernel involves five distinct arithmetic operations over the same input data. On the other hand, Matrix Multiplication, despite being $100\,\%$ parallelizable, only requires a single arithmetic operation per pair of operands (and a final reduc-

Table 4.1: Set of seven benchmarks from three different application domains used for evaluation. The used implementations of k-Nearest Neighbors (kNN), K-Means, Streamcluster, and Backprop are from the Rodinia benchmark suite [109].

Benchmark	Domain	Approx. accel. kernel time [%]
Matrix Add.	Algebra	$100\,\%$
Matrix Mul.		$100\,\%$
K-Nearest Neighbors	Clustering	$90\,\%$
K-Means		$95\,\%$
Streamcluster		$97\,\%$
YoloV3 [110, 111]	Machine Learning	$90\,\%$
Backprop		$80\,\%$

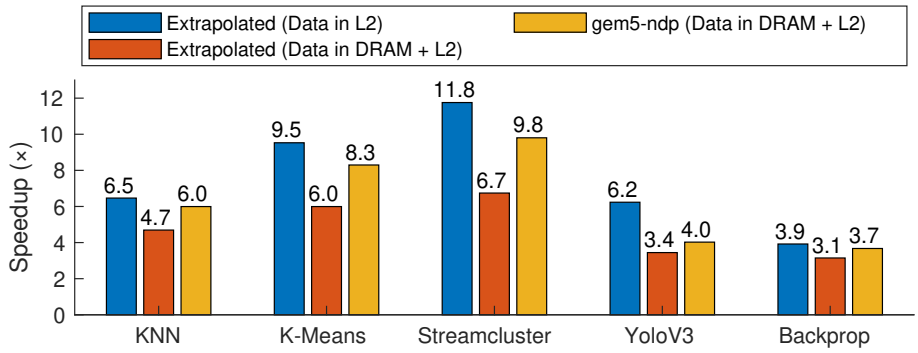


Figure 4.17: Performance benefits provided by the bidimensional NDP array considering: (blue and orange bars) the CPU and the NDAcc operating separately (extrapolated results); and (yellow bars) using NDPmulator. In this experiment, the NDAcc was connected to the L2 cache, the CPU and the NDAcc operate simultaneously, and the data is present in both the DRAM and the L2 cache.

tion). Thus, when considering the OoO CPU baseline and compilation with -O3 (which includes vectorization), the performance benefits attained by the NDAcc for the Streamcluster benchmark are higher than those for Matrix Multiplication, since it can be efficiently executed by the CPU. As expected, the main advantage of NDP over traditional CPU-only processing is its higher bandwidth to the memory. Adding multiple cores may increase the CPU compute capabilities, but its limited memory bandwidth does not allow to fully exploit such capabilities.

4.4.2.C Comparison with a Classical Evaluation Approach

The performance speedups obtained with NDPmulator were compared with the corresponding results when considering the absence of an integrated simulator. In such case, to estimate the performance, one would consider the execution times of the CPU and the NDAcc operating separately disregarding the access contention in the memory hierarchy. Hence, the Amdahl's law can be used by considering that the global speedup (S) can be obtained from the speedup of the workload accelerated by the NDAcc (s), and the proportion of execution time that the accelerated workload originally occupied (A) on the CPU.

$$S = \frac{1}{(1 - A) + A/s}$$

Figure 4.17 illustrates the performance benefits provided by the NDAcc obtained through: (1) extrapolation, considering that the operands of the NDP kernel are present in the L2 cache; (2) extrapolation, considering that the operands of the NDP kernel are distributed across the DRAM and the L2 cache with a realistic L2 hit rate obtained through simulation; (3) NDPmulator, with the operands being distributed across the DDR and the L2 cache, the NDAcc coupled to the L2 cache, and both the CPU and the NDP operating simultaneously, which corresponds to an

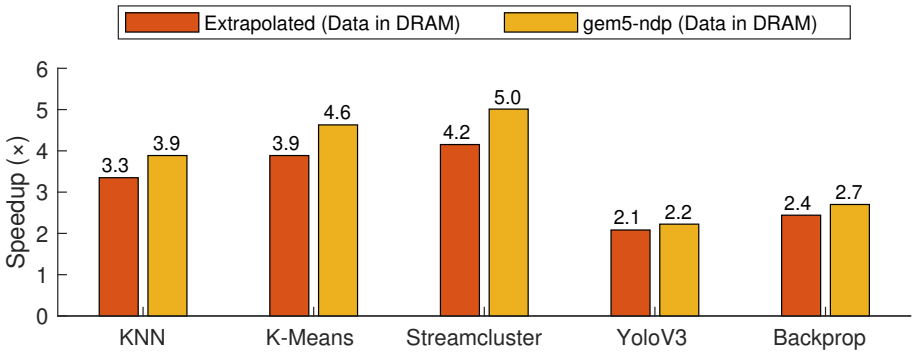


Figure 4.18: Performance benefits provided by the bidimensional NDP array considering: (orange bars) the CPU and the NDAcc operating separately (extrapolated results); and (yellow bars) using NDPmulator. In this experiment, the NDAcc was connected to the DRAM, the CPU and the NDAcc operate simultaneously, and the data present in the DRAM.

accurate simulation of the actual system. In addition, Figure 4.18 shows a similar evaluation where the NDAcc is coupled to the DDR memory.

As it can be observed, extrapolating the performance from the execution times of the CPU and the NDP alone is only useful to obtain the order of magnitude of the actual performance benefits, with errors ranging from $6.4\,\%$ to $54.9\,\%$. Even the results considering a realistic L2 hit rate incur in errors as high as $31.2\,\%$, which may be prohibitive when evaluating the viability of new NDP architectures. It is also noticeable that the errors are significantly lower when assuming the NDAcc connected directly to the DDR memory comparing to when the NDAcc is connected to the L2 memory. This effect can be explained by the fact that the latency of DDR accesses is much easier to predict than the average latency of the L2 cache, which is affected by the hit rate for a given data access pattern and the traffic generated by the CPU.

4.5 Experimental Evaluation: Modelling Existing NDAccs

To further validate and demonstrate the benefits of NDPmulator, three NDAcc architectures recently proposed in the literature were modeled using the information provided in the corresponding manuscripts. Then, a subset of the benchmarks originally used to evaluate those NDAccs were executed and the results compared with those reported by their authors.

4.5.1 Near-Data Database Processing

The first modeled NDAcc was proposed by Das and Kapoor [112] consisting in a Near-Data Compare Unit (NDCU) targeting the execution of common operations in Database (DB) applications. This NDAcc architecture is illustrated in Figure 4.19. Specifically, the proposed NDCU has the capability of scanning column-store DBs (the values of the same attribute are stored

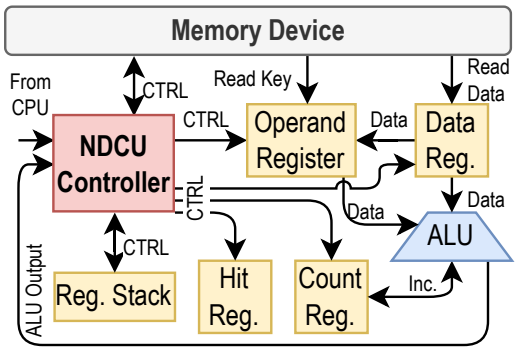


Figure 4.19: NDCU architecture proposed by Das and Kapoor [112].

sequentially in memory rows) and row-store DBs (the attributes of an entry are stored sequentially in memory rows), and implements the DB operations compare-n-hit, compare-n-count, and compare-n-max. The compare-n-hit operation scans the values of an attribute and sets the hit register only if a specified value is found; compare-n-count counts the number of occurrences of a value among all values of a given attribute; and compare-n-max scans the values of an attribute and determines its maximum value. Depending on the operation code sent by the CPU, the NDCU executes one of the three operations described above.

The authors placed their NDAcc in the logic layer of one or more vaults of an Hybrid Memory Cube (HMC), allowing to (1) directly fetch the operands from the stacked DRAM layers of each vault, (2) process the data, and (3) store back the results in the HMC.

Since the goal of this experiment is to demonstrate that NDPmulator allows to simulate NDAccs attached to any level of the memory hierarchy, it was decided to evaluate the devised NDCU when coupled to the L2 data cache. In addition, to facilitate the comparison between the results presented by Das and Kapoor [112] and those obtained using NDPmulator, the conducted evaluation focus on the column-store DB benchmarks using a single NDCU.

Table 4.2: Parameters used to configure the baseline system for the evaluation of the NDCU proposed in [112] and executed benchmarks.

System Configuration	
CPU	Single-core, x86-64, $2\,\mathrm{GHz}$
L1 i-cache	$32\,\mathrm{kB}$, 4-way, $64\,\mathrm{B}$ line
L1 d-cache	$32\,\mathrm{kB}$, 8-way, $64\,\mathrm{B}$ line
L2 cache	$256\,\mathrm{kB}$, 16-way, $64\,\mathrm{B}$ line
Main memory	DDR3 $1600\,\mathrm{MHz}$
Executed Benchmarks	
`hit best`	Searches a DB attribute column for a specific value and returns as soon as it is found terminating the search
`hit avg`	
`hit worst`	
`count`	Counts the number of occurrences of an attribute
`max`	Determines the maximum value of an attribute

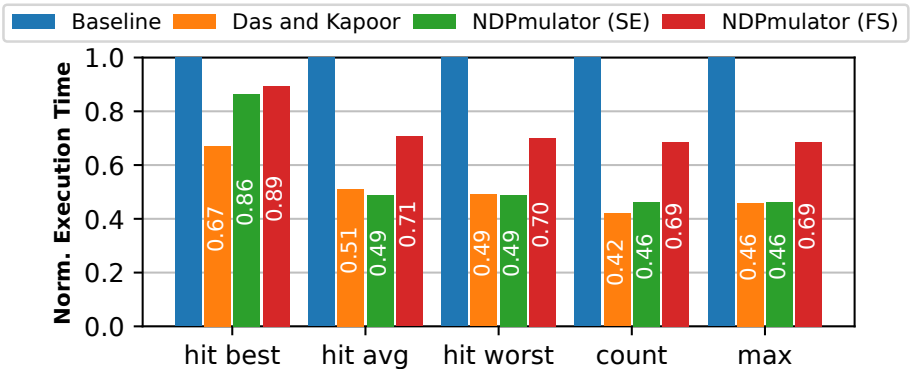
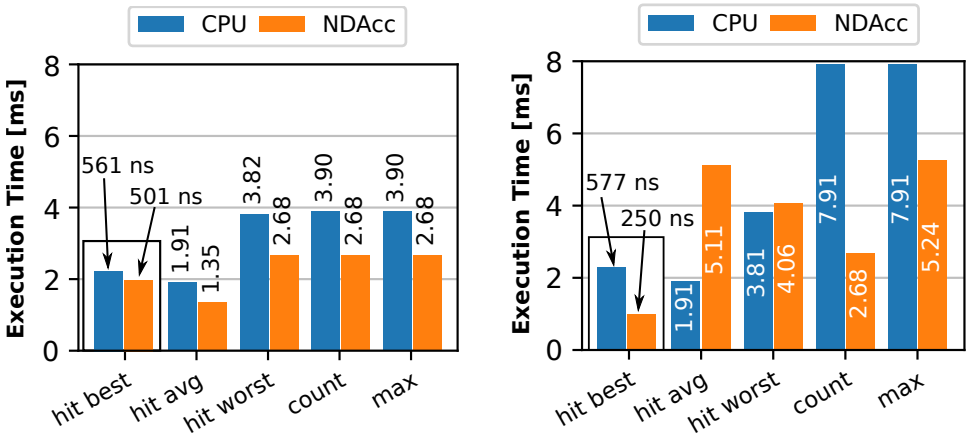


Figure 4.20: Performance benefits of the NDCU presented in [112]. The second bar represents the results obtained by Das and Kapoor [112] in their original evaluation. The third bar illustrates the estimated performance benefits of the NDCU modeled using NDPmulator when coupled to the L2 data cache in SE mode. Finally, the last bar depicts the performance benefits estimated using the FS mode of NDPmulator with the NDCU also coupled with the L2 cache. The results are presented in the same format as in the original paper [112].

To evaluate this NDAcc using NDPmulator, a total of four evaluation scenarios were considered and, for each scenario, five benchmarks extracted from the original paper [112] were executed. In the first evaluation scenario, the SE mode was used with the NDCU connected to the L2 data cache. The remaining three evaluation scenarios used FS mode with the NDCU also coupled with the L2 data cache. These benchmarks were executed (1) with no other process running, (2) with another data-intensive workload being simultaneously executed in the CPU, and (3) using a device driver to intermediate the control of the NDCU via the OS kernel, while no other process was being executed. The five benchmarks used to evaluate the NDCU are summarized in Table 4.2 and all represent their data with 64-bit unsigned integers. It is also worth mentioning that the baseline used for both the SE and FS scenarios was the same used by Das and Kapoor [112] in their original work, as shown in Table 4.2. Furthermore, all tests were made using the x86 ISA. For the FS mode setup, a recent Linux kernel (5.4.0-105) was used, together with the Ubuntu 18.04 root file system.

For convenience, it is assumed that all datasets used by the performed tests have $64\,\mathrm{kB}$. Although this might result in small benchmarks, it is worth noting that it is not the goal of this paper to validate the NDAcc proposed in [112], but to demonstrate the potential or capability of the proposed framework to reproduce and anticipate the results of such NDAccs. In particular, the use of smaller benchmarks is herein more interesting for highlighting the OS-related overheads, which are only observable when simulating in FS mode.

To facilitate the analysis of the results, Figs. 4.20 and 4.21 consider scenarios where the benchmarks executed in FS mode did not include any intermediary device driver. The commu-



(a) Exclusive execution

(b) Concurrent execution

Figure 4.21: Execution time of the five benchmarks used to evaluate the NDCU proposed by Das and Kapoor [112] obtained with NDPmulator FS mode. In (a), no other workload is being executed in parallel with the considered benchmarks. (b) illustrates a scenario where another data-intensive workload is being executed in parallel with the benchmarks. Due to its much smaller execution time, `hit best` results were represented using a different scale in the vertical axis.

nication between the CPU and the NDAcc was implemented by directly mapping the PI registers into the application addressing space (requiring superuser privileges) using the Linux system call `mmap`. Therefore, these scenarios do not consider the communication overhead introduced by the device driver. On the other hand, the results shown in Figure 4.22 specifically target the quantification of the overhead introduced by the existence of the provided device driver.

Figure 4.20 presents the NDPmulator performance when simulating the same benchmarks used by Das and Kapoor [112]. In particular, the third bar in each set (green) was obtained using SE mode with the NDAcc coupled to the L2 cache. As it can be observed, the results obtained using the SE mode are very similar to those obtained in the original evaluation of the circuit presented by Das and Kapoor [112], but requiring a rather smaller development cost to attain such estimation of performance. Moreover, the author's original results were obtained considering NDP on an HMC, whereas we assess the performance of an equivalent system implemented near an L2 cache.

The last bar (red) considers a similar scenario except that FS mode is used. While significant performance benefits are still achieved (when compared with the baseline), the overall performance of the NDAcc is smaller. This can be explained by the overhead introduced by the OS. In particular, the OS overheads are lower for the `hit best` benchmark due to the reduced influence of the scheduler preemption mechanisms as the execution time is substantially lower. On the other hand, for the other benchmarks, the preemption mechanisms and the sparse and irregular memory accesses made by other background applications may introduce contention on shared

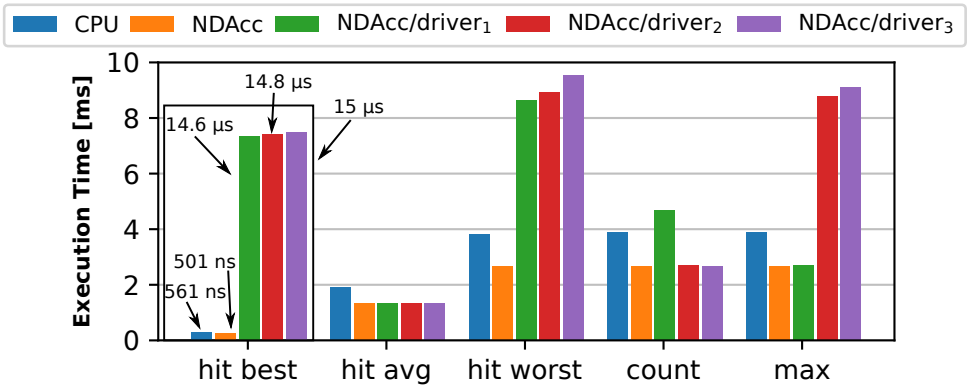


Figure 4.22: Execution time of the five benchmarks used to evaluate the NDCU proposed in [112] considering: only the CPU; the CPU plus the NDAcc being controlled by simply and directly reading and writing from/to its programming registers; the CPU and the NDAcc being controlled through a proper device driver (bars 3–5, each corresponding to a different run). Due to its much smaller execution time, `hit best` results were represented using a different scale in the vertical axis.

resources, increasing the overall entropy at memory level and influencing the maximum data rate that can be achieved by the NDAcc. It is also worth noting that this effect is even more evident due to the rather short execution time of the benchmarks ($\sim 10\,\mathrm{ms}$).

The performance variations due to simultaneous accesses to the memory hierarchy are yet more visible in Figure 4.21, where (a) shows the execution time of the CPU and the NDAcc when no other process is being executed in the CPU and (b) illustrates a scenario where another data-intensive workload is concurrently executing in the CPU, leading to simultaneous memory accesses that cause contention in the access to the shared resources and also pollutes data caches. It is worth mentioning that this scenario can only be evaluated by simulating the entire processing system, including the NDAcc and an OS running on top of the hardware infrastructure, in a real multi-tasking environment. To our knowledge, this is an exclusive feature of the FS mode provided by NDPmulator.

To conclude this evaluation, a device driver was developed to intermediate the communication between the CPU and the NDAcc using the provided OS security and isolation mechanisms. The corresponding results are illustrated in Figure 4.22.

Although interfacing an application with the NDAcc through a device driver brings significant benefits in terms of security, it also has important implications in terms of performance. In this experiment, the synchronization points where the user application communicated with the NDAcc by directly accessing its PI registers were replaced by calls to the device driver, without modifying the structure of the code. The presented results show that the use of the device driver results in an overhead of up to $7\,\mathrm{ms}$, which impact on the overall system performance, depending on

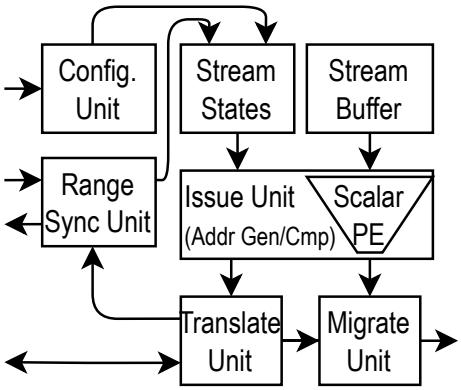


Figure 4.23: NDAcc architecture proposed by Wang *et al.* [113, 114].

the execution time of each issued NDAcc kernel. Naturally, since the execution time of the `hit best` benchmark is lower, the driver communication overhead leads to a higher relative increase in execution time.

Furthermore, the device-driver-based communication implemented between the CPU and the NDAcc is highly dependent on the underlying OS mechanisms, leading to significant performance variations even when executing the same workload several times. This effect can be seen in the last three bars of Figure 4.22, which correspond to the exact same workloads (using the device driver) and yet produce results with a maximum difference of $6.4\,\mathrm{ms}$.

4.5.2 Near-Stream Computing

The second use case that was used to further corroborate the functionality of the proposed NDPmulator framework was the NDAcc presented by Wang *et al.* [113, 114]. Their architecture consists of PE arrays installed close to the cache to perform arithmetic and logic vector operations, as depicted in Figure 4.23. Their processing structure not only allows to take greater advantage of the memory devices bandwidth at that level of the hierarchy to increase the exploited parallelism but also allows the processing of data in a streamed manner, reducing the overall latency of operations. Furthermore, it includes a compiler solution to automatically make use of the considered hardware infrastructure, thus taking full advantage of its performance benefits.

In [113], the authors attached their NDAcc close to the L3 data cache and assessed its performance benefits using a set of eight benchmarks. These same benchmarks were also used to evaluate this NDAcc architecture with the NDPmulator framework (see Table 4.3). Due to the unavailability of the compiler tool-chain used by Wang *et al.* (as well as the used implementation of the executed benchmarks), other common reference implementations of the said applications were used and modified to include explicit calls to the NDAcc. Furthermore, due to the unavail-

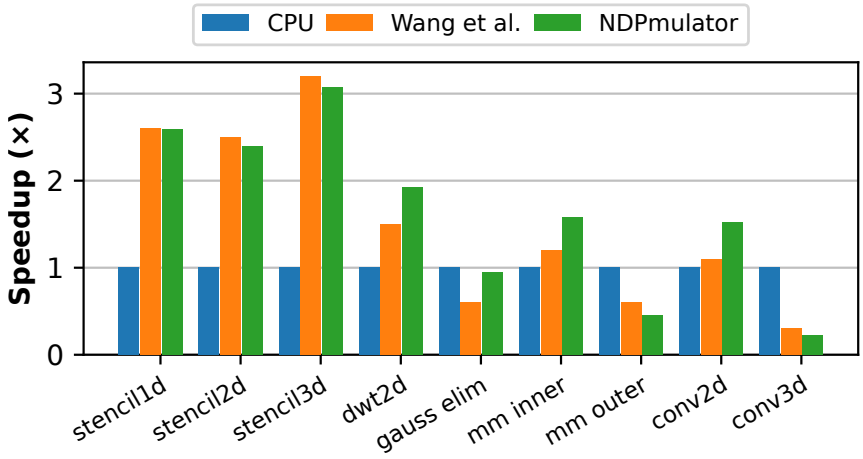


Figure 4.24: Performance benefits of the NDAcc proposed by Wang *et al.* [113, 114] compared with those estimated using NDPmulator.

ability of the same exact components used in the gem5 evaluation presented by Wang *et al.*, the authors attempted to approximate (as much as possible) the original evaluation system using existing gem5 modules, as described in Table 4.3. Nevertheless, even with these restrictions, NDPmulator still allowed to approximate the original testing setup and obtain fairly accurate performance results, with three out of the eight tested benchmarks presenting an error below $5\,\%$, as shown in Figure 4.24. All results regarding this use case were obtained using the SE mode. The data type used with this NDAcc was single-precision floating point (contrasting with 64-bit unsigned integers used with the architecture of Das and Kapoor [112]), showing that NDPmulator supports any data type used by the NDAcc.

Table 4.3: Parameters used to configure the baseline system for the evaluation of the NDAcc proposed in [113, 114] and executed benchmarks.

System Configuration	
CPU	Single-core, x86-64, $2\,\mathrm{GHz}$
L1 i-/d-cache	$32\,\mathrm{kB}$, 8-way, $64\,\mathrm{B}$ line
L2 cache	$256\,\mathrm{kB}$, 16-way, $64\,\mathrm{B}$ line
L3 cache	$256\,\mathrm{MB}$, 16-way, $64\,\mathrm{B}$ line
Main memory	DDR4 $3200\,\mathrm{MHz}$
Executed Benchmarks	
`stencil1d`	1D, 2D, and 3D stencil algorithm—used to approximate the solution of partial differential equations based on a discrete set of values
`stencil2d`	
`stencil3d`	
`dwt2d`	2D Discrete Wavelet Transform
`gauss elim`	Gaussian elimination algorithm
`mm inner`	Matrix Multiplication (inner product)
`mm outer`	Matrix Multiplication (outer product)
`conv2d`	2D and 3D convolution kernels—the most used operation in CNNs
`conv3d`	

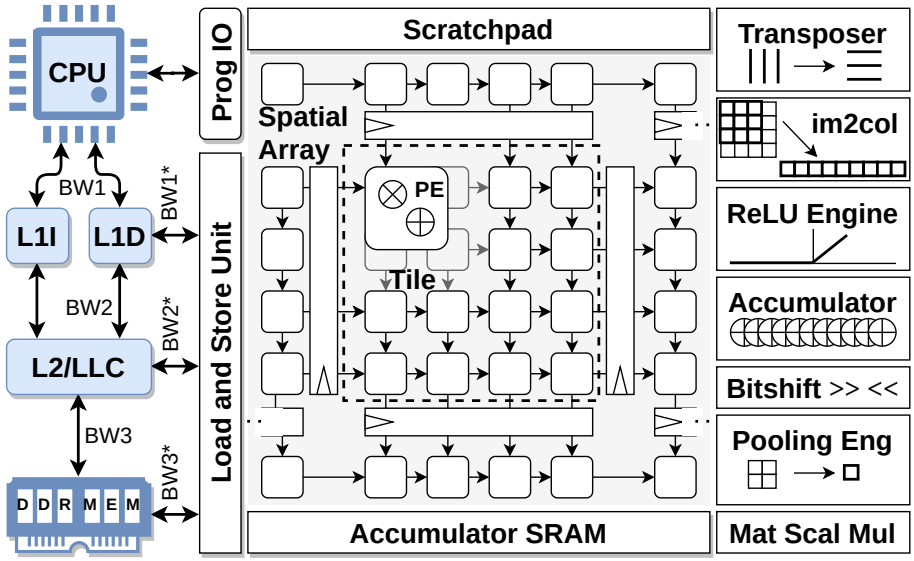


Figure 4.25: Generic architecture of a Gemmini NN accelerator consisting of a configurable systolic mesh of PEs grouped into tiles, as well as NN-specific units and input and output memories.

4.5.3 Gemmini Accelerator

The third and final NDAcc used to validate NDPmulator was inspired in Gemmini [115], an architectural framework to create accelerators targeting the execution of Neural Networks (NNs). The considered NDAcc (see Figure 4.25) consists of a bi-dimensional systolic array of PEs, with each PE containing a Multiply-Accumulate (MAC) unit and communicating with its neighboring PEs through dedicated buses. Additionally, PEs can be arranged into tiles, which communicate with adjacent tiles through pipeline registers.

Gemmini accelerators may also include a specialized unit to perform matrix transposition, an engine to rearrange the entries of input matrices for convolution, a Rectified Linear Unit (ReLU), an accumulator, a bit-shift unit, a pooling engine, and a matrix-scalar multiplier. In addition, an input memory (scratchpad) and an output memory (accumulator SRAM) are also included. Both these memories are explicitly managed, i.e., the CPU is responsible for explicitly transferring the operands to the input memory and retrieving the results from the output memory.

In the conducted experiments, the modeled Gemmini NDAcc consists of a systolic mesh of 16-by-16 PEs with a $256\,\mathrm{kB}$ scratchpad memory and a $64\,\mathrm{kB}$ accumulator output memory, with the remaining processing system being as described in Table 4.4.

Three experiments were performed using the aforementioned setup (see Table 4.4): (1) five microbenchmarks consisting of common NN operations, to be executed using the Gemmini accelerator; (2) twelve Convolutional Neural Network (CNN) models to be run using a modified version of the Darknet [116] framework executed on the Gemmini accelerator (inference only); and (3)

three hand-tuned CNNs to be executed using the described hardware configuration to fully exploit the benefits of the modeled Gemmini accelerator. In addition, the nine microbenchmarks of experiment (1) were also executed using an official Gemmini emulation platform supported on Verilator and the results were compared with those obtained with NDPmulator. For simplicity, the experiments were conducted using SE mode. Nevertheless, NDPmulator also offers full support to FS mode.

Figure 4.26a and Figure 4.26b illustrate the results obtained for the first experiment. The considered microbenchmarks are divided into two groups: (a) kernels whose operands are first gathered by the CPU and stored in the scratchpad, in the order they are required for the operations; and (b) kernels that are fully executed by the NDAcc. In the first scenario, the CPU replaces the functionality of the `im2col` and `transposer` blocks of the Gemmini NDAcc. Naturally, this may lead to a larger memory footprint in the scratchpad (because the convolution operands are stored in a redundant form), and to a significant increase in execution time (since not only more data may be transferred from the memory hierarchy to the scratchpad but the performed accesses are also irregular). Hence, the benchmarks belonging to the first group attain a much lower speedup than their counterparts of the second group, which are exclusively executed by the Gemmini NDAcc, including the gathering of the operands.

Nevertheless, existing applications meant to be executed by the CPU/GPU often gather the operands beforehand, organizing them in memory in a computation-friendly manner (similarly to the kernels of Figure 4.26a). However, offloading that process to the implemented NDAcc would require complex structural modifications to the applications. On the other hand, the computation itself can still be moved from the CPU/GPU to the NDAcc with simple changes to the code.

Table 4.4: Parameters used to configure the baseline system for the evaluation of the NDAcc proposed in [115] and executed benchmarks.

System Configuration	
CPU	Single-core, x86-64, $2\,\mathrm{GHz}$
L1 i-/d-cache	$32\,\mathrm{kB}$, 8-way, $64\,\mathrm{B}$ line
L2 cache	$256\,\mathrm{kB}$, 8-way, $64\,\mathrm{B}$ line
Main memory	DDR3 $1600\,\mathrm{MHz}$
Executed Benchmarks	
`conv2d`	2D and 3D convolution kernels—the most used operation in CNNs
`conv3d`	
`maxpool`	Max pooling kernel
`relu`	Rectified Linear Unit kernel
`mm`	Matrix Multiplication
Full CNN benchmarks	12 darknet CNN models and 3 hand-tunned CNN models

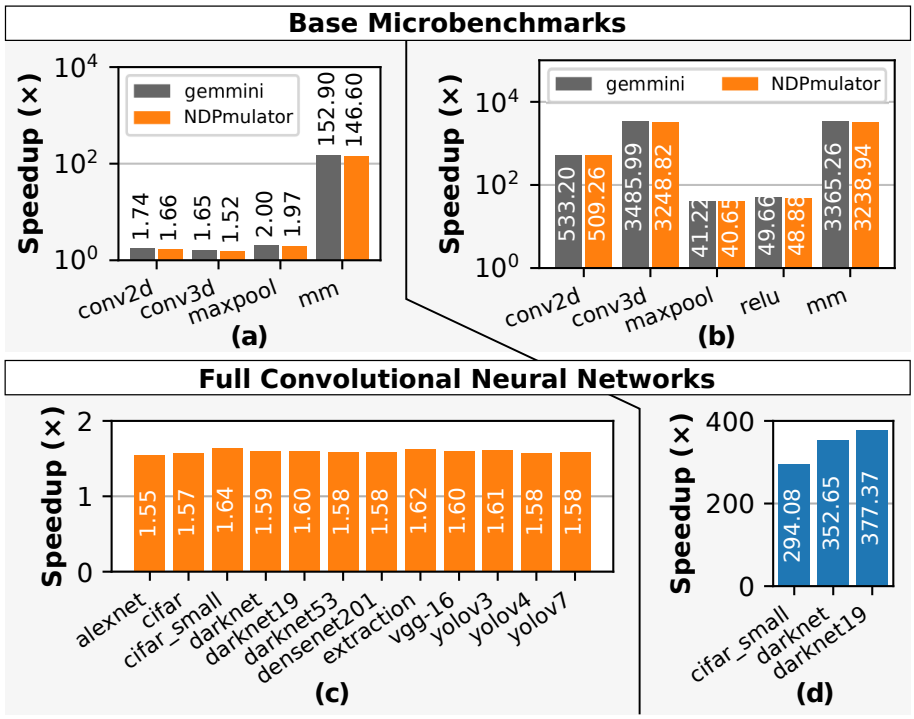


Figure 4.26: Microbenchmarks (a, b) and full CNN benchmarks (c, d). (a) and (c) show the results for applications where the gathering of the operands is done by the CPU. (b) and (d) correspond to applications fully executed by the Gemmini NDAcc.

Therefore, Figure 4.26a essentially illustrates the speedup attainable by legacy kernels optimized for execution in CPU/GPU that were later adapted to be executed in the NDAcc apart from the gathering of the operands.

When comparing the performance results of NDPmulator (orange bins) with those of Gemmini's official simulation platform (gray bins), an average error as low as $4.3\,\%$ can be observed, supporting the accuracy of the devised model.

Furthermore, when comparing the performance benefits allowed by the Gemmini NDAcc for the matrix multiplication benchmark with those observed with the bidimensional NDP array in section 4.4.2, it can be observed that the Gemmini NDAcc performs significantly better than the bidimensional NDP array. Since the data paths of these two accelerators are very similar (for this specific benchmark), their performance discrepancy can be explained by the fact that, in contrast with the bidimensional NDP array, the Gemmini NDAcc uses direct address mapping. This does not only guarantee that consecutive virtual addresses correspond to consecutive physical addresses in memory (i.e., data is not scattered in memory, which allows fetching operands and storing results more efficiently) but also avoids the expensive process of explicitly translating virtual addresses into their physical counterparts using a secondary TLB and the MMU.

A second experiment using the Gemmini NDAcc considers an evaluation using complete CNN

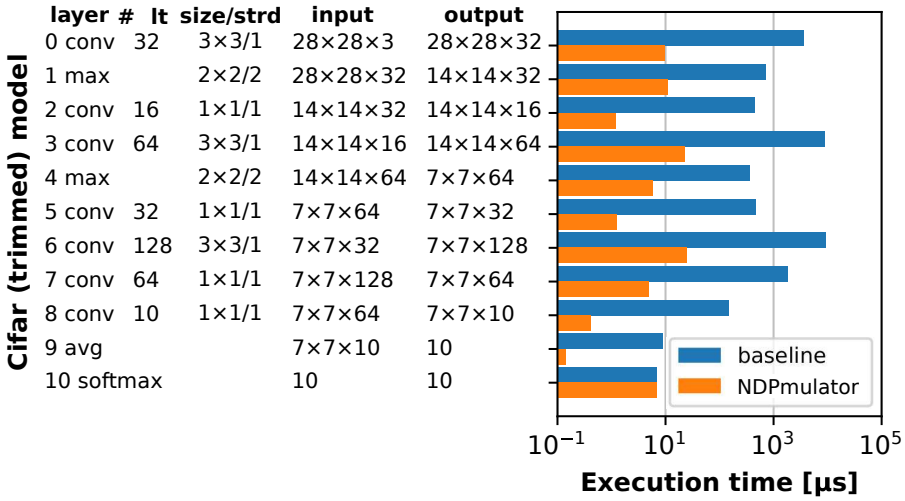


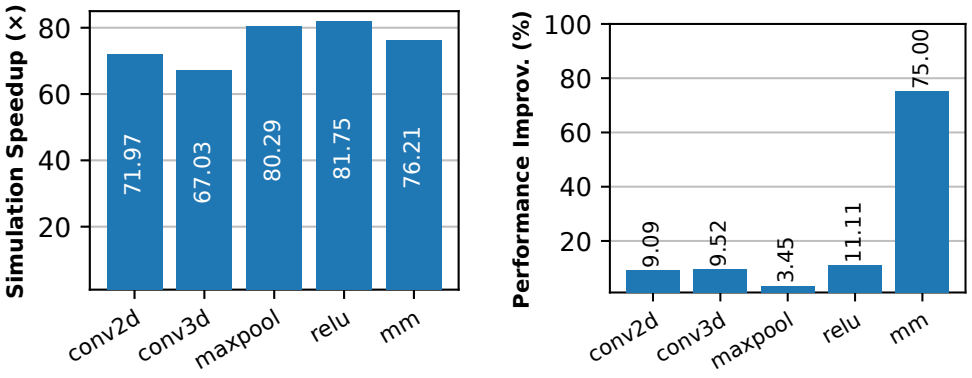
Figure 4.27: Execution time of each layer of a small CNN optimized for the CIFAR-10 dataset when executed only by the CPU (blue bars) and the CPU equipped with the NDAcc (orange bars).

models (through the Darknet framework). However, due to the way Darknet is organized (which benefits the execution in CPUs and GPUs), offloading the gathering of the operands to the NDAcc would require rather complex structural modifications. Thus, in the second experiment, only the convolution and pooling operations were completely offloaded to the Gemmini NDAcc, while the gathering of the operands remained being executed by the processor. Not surprisingly, this led to much lower performance benefits, with speedups ranging from 1.55× to 1.64× across the twelve tested CNNs (inference only), as shown in Figure 4.26c.

Nevertheless, this experiment shows the robustness of NDPmulator, allowing to execute a benchmark as complex as Darknet in a realistic processing system featuring a custom NDAcc together with a communication layer between the host code and the NDAcc.

Figure 4.26d shows the speedup obtained when executing three hand-tuned CNNs to evaluate the use of the full capabilities of the modeled NDAcc, particularly its capability to directly process the data in memory. By offloading the gathering of the data to the NDAcc, i.e., using the `im2col` and `transposer` blocks, speedups of more than two orders-of-magnitude are achieved, similarly to the results presented in [115]. In particular, Figure 4.27 depicts the decrease of the execution time of convolutional and pooling layers for a small CNN targeting the CIFAR-10 dataset.

Finally, two additional tests were performed to demonstrate the simulation performance benefits and the versatility of NDPmulator. Figure 4.28a depicts the simulation speedup of NDPmulator over an official Verilator-based Gemmini simulation platform. As it can be observed, NDPmulator allows for simulation speedups as high as 81.75×, while still being able to accurately execute the considered benchmarks.



(a) Simulation performance benefits

(b) Benefits of 32 B over 16 B bus

Figure 4.28: (a) Simulation speedup of NDPmulator over an official Gemmini simulation platform. (b) Performance benefits obtained when increasing the width of the bus connecting the Gemmini NDAcc with the L2 cache, which can be determined by simply changing a parameter in the simulation file.

On the other hand, NDPmulator allows to easily study the impact of changing the parameters of the system. For example, Figure 4.28b illustrates the performance benefits of a system featuring the Gemmini NDAcc connected to the L2 data cache through a $32\,\mathrm{B}$ bus over a similar system where the Gemmini NDAcc is connected to a $16\,\mathrm{B}$ crossbar. These results are quite expected, with the performance almost doubling in the case of the matrix multiplication benchmark due to the increased bandwidth to the memory device. The benefits observed for the remaining benchmarks are quite lower, which is explained by their stridden memory access patterns (which reduces the usable bandwidth as only a few words of a cache line are used).

4.6 Summary

In this chapter, a novel simulation toolchain targeting NDAccs called NDPmulator is proposed. NDPmulator is a gem5-based full-system solution to develop, test, integrate, and validate new NDAccs, without resourcing to complex RTL development approaches. The proposed toolchain not only allows the simulation of the simultaneous operation of the CPU and NDAccs but also supports the integration of (multiple) NDAccs at any level of the memory hierarchy or even multiple levels. In addition, NDPmulator provides custom simulation scripts, allowing the evaluation of NDAccs by executing a single NDP-enabled application in isolation (SE mode) or by simulating a fully-featured operating system running on top of the hardware infrastructure (FS mode).

NDPmulator is compatible with all ISAs supported by gem5 in SE mode, and with x86, ARM, and RISC-V ISAs in FS mode. To our knowledge, NDPmulator is the first proposal consisting of an integrated full-system development and simulation environment for NDAccs. All other known

solutions require the use of multiple tools, post-simulation steps, repurpose tools not tuned for NDP evaluation, are limited to a few NDAcc architectures, and do not provide OS support.

As a simulation toolchain, NDPmulator naturally operates over models that have some degree of abstraction when compared with the actual physical hardware they represent. Therefore, it is not capable of producing RTL code nor guaranteeing that the simulated model has a physical equivalent. In other words, while NDPmulator is capable of great rigor when provided with a detailed well-parameterized model of an NDAcc, if the supplied model is not realistic, the produced results will not be accurate. Hence, it becomes the responsibility of the programmer to correctly model and parameterize the NDAcc under development.

Nevertheless, NDPmulator still allows to correlate the high-level model of the accelerator and the corresponding physical hardware (electronic components). That specification can then be delivered to McPAT [52] and CACTI [96] together with simulation traces to estimate the circuit's area and energy requirements (taking advantage of the already existing hardware and energy profiles of the gem5 components).

Two case studies were presented to illustrate the process of developing and evaluating new NDAccs using NDPmulator, as well as to show the benefits of using the framework compared to extrapolating performance gains from the execution time of the CPU and NDAccs. This exercise allowed to conclude that neglecting the influence of the surrounding processing system (i.e., considering only the CPU and NDAccs alone for evaluation purposes) may incur significant errors when estimating the attained performance benefits. In the performed experiments, those errors were as high as $54.9\,\%$. On the other hand, by simulating the entire processing system, NDPmulator eliminates those limitations, accurately predicting the real gains provided by NDAccs.

In addition, three existing NDAccs from the literature were modeled and evaluated using NDPmulator. The obtained experimental results show that the proposed toolchain allows estimating the results obtained by the original authors without the need to formally implement the corresponding architectures, in a fraction of the development time. In addition, by allowing the simulation of an entire processing system featuring NDAccs and even an OS being executed on top of the simulated hardware, accurate results that would otherwise be impossible to obtain can be extracted. Furthermore, NDPmulator offers significant simulation performance improvements over alternative evaluation platforms, being capable of estimating the performance of NDAccs in a fraction of the time. Moreover, NDPmulator allows to easily evaluate the impact of changing core features of a system by simply changing few parameters, while classic evaluation strategies would require significantly more effort if not a partial redesign of the entire system.

Finally, NDPmulator is an open-source tool publicly available at `https://github.com/hpc-ulisboa/NDPmulator`. Instructions on how to obtain the FS file system images and kernels are also available at this location.

5

Conclusions

Contents

Driven by the need to mitigate the gap between the computing power of recent processing hardware and the insufficient performance of current memory subsystems, the Near-Data Processing (NDP) paradigm aims at moving the computation closer to the data storage location, and, at the same time, exploiting higher bandwidth and parallelism opportunities, which are hardly available with conventional Central Processing Units (CPUs) due to interconnection contention.

Throughout the years, several NDP approaches have risen, with the first proposals being mostly based on Processing in Memory (PIM) using Dynamic Random-Access Memory (DRAM) arrays. More recently, this paradigm was also adapted to Static Random-Access Memory (SRAM) and Resistive Random-Access Memory (RRAM). However, classic PIM still suffers from complex drawbacks, such as the design constraints imposed by the internal architectures of the memory devices, the highly heterogeneous nature of the computation (which makes it difficult to create a standard programming paradigm), and the cost of implementation (as most PIM architectures require the redesign of the memory chips).

To mitigate this, an alternative NDP concept called Processing Near Memory (PNM) was created, which proposes to embed the computational hardware close to memory devices (rather than inside the memory arrays). PNM has the advantage of allowing a much richer design exploration, as the architectures of the Near-Data Accelerators (NDAccs) are not as dependent on the internal architecture of the memory devices. Contrasting with PIM devices, which rely exclusively in the existing internal memory resources to perform computation and require specific data placements, PNM devices are much less reliant on the physical architecture of the memory and the data placement, as they are located outside the memory controller. Plus, while PIM devices can only perform rather simple logic instructions (often requiring thousands of cycles to implement more complex arithmetic operations), PNM devices make use of standard digital Functional Units (FUs) to efficiently implement complex arithmetic instructions.

While traditional PIM devices are constrained by the internal memory resources (e.g., DRAM-based PIM, which uses bit-line computation, relies on the physical placement of data in the DRAM cells and the capabilities of the existing sense amplifiers to perform computation) and can only perform rather simple logic instructions (often requiring thousands of cycles to implement more complex arithmetic operations), PNM devices make use of standard digital FUs, and are much less dependant on data placement inside the memory devices (as they are located outside the memory controller).

Furthermore, this also makes it easier to devise generic programming paradigms, as the physical architecture of the memory devices is abstracted by the memory controller. Therefore, the in-

ternal architecture of the memory is not as present in the programming of PNM devices as in PIM devices. Moreover, in the past decade, two memory technologies were proposed that combine memory dies and logic dies within the same chip–the Hybrid Memory Cube (HMC) and the High-Bandwidth Memory (HBM)–, making it possible to implement fully digital PNM devices alongside memory devices within the same chip.

Leveraging the benefits of NDP, as well as cache locality and parallelization opportunities enabled by important signal-processing applications, a new PNM architecture was developed specifically aimed at signal processing. The devised Cache Compute System (CCS) was integrated with a standard processing system featuring a high-performance ARM Cortex-A53 CPU with a two-level cache hierarchy and a DDR main memory, with the CCS being coupled to the Last-Level Cache (LLC). The system equipped with the devised CCS showed significant performance benefits, with speedups ranging from 3.9× to 40.6×. However, the development of the CCS uncovered a critical aspect in the field of NDP: the scarcity of tools to support the development and evaluation of NDAccs.

While NDAccs are hardware accelerators in their essence, their close coupling with the CPU (for control purposes) and the memory devices (for transferring data at high speeds between the memory arrays and the processing units) make their evaluation crucially different from standard accelerators/co-processors. In particular, when evaluated in separate, results tend to be too optimistic, as CPU synchronization overheads and memory contention are not taken into account. On the other hand, evaluating NDAccs in an integrated environment is a complex task and classic Register Transfer Level (RTL) development methods are expensive and time-consuming.

Over the past decade, significant strides have been made to address the challenges in designing pre-RTL development tools for heterogeneous high-performance processing systems. These efforts highlight critical issues such as the inadequacy of accurate models and the scalability limitations of existing solutions [4, 5, 67, 100–102]. Although notable advances in design exploration and performance prediction have been made, these typically focus on relatively homogeneous systems, like conventional multi-core CPU systems [103–105], and are not well-suited for heterogeneous systems that include specialized hardware accelerators. Moreover, they are inadequate for early design exploration phases where the architectural parameters are not yet fixed, as they often require a detailed specification of the device [17].

Additionally, many artifacts used in current research are either not optimized for evaluating NDAccs (e.g., in [15], the authors utilize a Graphics Processing Unit (GPU) simulator to assess their NDP architecture), are restricted to a limited number of architectures [18], or rely on post-

simulation steps to integrate the performance metrics of the NDAccs with the rest of the system. This reliance results in the inability to simulate the concurrent operation of the CPU and the NDAcc, complicating the comparison of evaluation methodologies and potentially questioning the validity of the results obtained.

Hence, it is fundamental to develop rigorous mechanisms dedicated to the validation and evaluation of new NDAccs without undergoing the expensive development procedures required by traditional full-scale methods, while still allowing to obtain accurate and trustworthy results.

To answer this need, the second part of this thesis focuses on the creation of a set of pre-RTL development tools specifically aimed at evaluating NDAccs, allowing to model, develop, simulate, and validate their operation at different levels of proximity to memory devices. The proposed toolchain [22, 23] is based on the widely established gem5 simulator [21], offering a full-stack development and evaluation solution for new NDAccs.

The proposed toolchain not only supports the simulation of individual applications using an NDAcc (System Emulation (SE) mode) but also provides the ability to execute several applications in parallel on top of an Operating System (OS) (Full System (FS) mode), allowing for a comprehensive evaluation of NDAccs under realistic circumstances, with accurate results. Furthermore, it supports all Intruction Set Architectures (ISAs) allowed by gem5 in SE mode and the x86, ARM, and RISC-V in FS mode. It also includes Linux device drivers to easily establish communication between a CPU application and the NDAcc under evaluation.

To our knowledge, the proposed toolchain is the first cycle-accurate simulation tool providing a full-stack solution specifically aimed at the development of NDAccs, covering not only the conception of the accelerator architecture but also its integration with the processing system and the co-execution of real NDP-enabled applications that make use of NDAccs.

All in all, the proposed toolchain offers the following unique compilation of features targeting the development of NDAccs:

- An open-source gem5-based simulation framework targeting the development and early evaluation of NDAccs, providing support for the development of their architectures and integration with the remaining processing system;
- Full support for simulating the simultaneous operation of the CPU and the NDAcc on a clock-cycle basis;
- Support for both SE and FS simulation, allowing to accurately evaluate NDAccs with a single application using an NDAcc or multiple processes on top of an OS;
- Native support for all ISAs supported by gem5 in SE mode, and support for x86, ARM, and

RISC-V in FS mode;

- Simulation scripts targeting both SE and FS modes;
- NDAcc Linux device drivers to be used in FS mode;
- Extensive validation of the toolchain, considering several distinct NDAccs from the literature, showing its versatility, accuracy, and simulation efficiency when compared with other state-of-the-art platforms.

In summary, this thesis aims at exploiting the potential of NDP techniques as an alternative to general-purpose CPUs to accelerate and optimize the execution of an ever-growing universe of data-centric applications from a wide range of scientific fields. In particular, a novel multi-purpose PNM architecture is proposed targeting prominent signal processing applications such as Convolutional Neural Networks (CNNs). In addition, this thesis also focus on the creation of tools to support the development of this class of devices, which are still scarce, aiming at contributing to the community's know-how to develop, validate, and evaluate NDP devices and encouraging the research of NDP solutions as an alternative high-performance computing paradigm. The valuable contributions of the work of this thesis were recognized by the community with the publication of five articles in prestigious international conferences and journals.

5.1 Future Work and Research Directions

In this section, future research directions regarding the work presented in this thesis are proposed. Namely, the study of a particular class of devices is envisaged: streaming accelerators. Streaming processing is crucial in high-performance computing because it enables the continuous and efficient handling of large volumes of data in real-time, significantly improving computational throughput and reducing latency. By processing data as it arrives, streaming accelerators can exploit parallelism and pipelining, which are essential for applications requiring real-time analytics, such as financial trading systems [117, 118]. This method also helps in managing the data flow between different stages of computation, ensuring optimal resource utilization and minimizing idle times. Studies have shown that streaming architectures can substantially enhance the performance of deep learning models and other data-intensive tasks by maintaining a steady flow of data through computational pipelines [119–124].

To further exploit the benefits provided by streaming processing, an interesting approach consists of installing streaming accelerators near the memory devices (i.e., combine streaming and near-data processing), allowing to exploit the high-bandwidth and low latency at that level, achieving higher data streaming throughput. For that purpose, the extension of NDPmulator [24] is en-

visaged to natively support the Unlimited Vector Extension (UVE) standard [125], as well as the corresponding compilation strategies proposed in [126].

Accordingly, the following tasks are regarded as a natural extension of the work of this thesis and in particular NDPmulator:

- **Extension of NDPmulator's architectural framework to natively support streamed memory accesses:** Entails the development of a streaming engine capable of interpreting UVE stream descriptors provided by the host CPU, calculating the corresponding memory addresses of the elements of the streams and prefetching the corresponding data from memory in an efficient manner. In particular, this task entails the specification of the communication protocol between the CPU and the streamed NDAccs to pass information regarding the UVE stream descriptors as well as the development of the architecture of the underlying hardware infrastructure responsible for managing the streams to be modeled.

- **Implementation of streaming NDP architectures for validation and assessment purposes:** Consists of the adaptation of known NDP device architectures such as [115] to support data streaming and coupling of such NDAccs to the previously mentioned streaming engine. Specifically, the memory accesses will be separated from the computation flow, making these architectures unaware of explicit memory accesses and capable of performing computation over a stream of data according to chronological instructions (i.e., instructions to be executed in specific clock cycles) precompiled by the CPU.

- **Portability of compiler-generated code for CPUs:** A tool will be developed to generate a configuration bitstream from the code generated by the existing compilation solutions for UVE. The resulting bitstream will be used be used by the implemented NDAccs for configuring the hardware Processing Elements (PEs) at run-time, implemented the desired functionality.

- **Validation of NDPmulator's streaming architectural framework:** The devised NDPmulator's streaming architectural framework will be validated and evaluated through the development of UVE-NDP compatible benchmarks and case studies.

The aforementioned work will be developed in collaboration and the context of project UNIFY (DOI: 10.54499/2022.06780.PTDC). The primary objective of this work is to publish a scientific article in a highly prestigious international conference, such as the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). This

collaboration aims to contribute significantly to the field, ensuring that the research meets the high standards required for acceptance at top-tier conferences.

Bibliography

[1] Xingqi Zou, Sheng Xu, Xiaoming Chen, Liang Yan, and Yinhe Han. Breaking the von neumann bottleneck: architecture-level processing-in-memory technology. Sci. China Inf. Sci., 64(6), 2021.

[2] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google workloads for consumer devices: Mitigating data movement bottlenecks. In ASPLOS, pages 316–331. ACM, 2018.

[3] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David T. Blaauw, and Reetuparna Das. Compute Caches. In HPCA, pages 481–492. IEEE Computer Society, 2017.

[4] Gabriel Falcão and João Dinis Ferreira. To pim or not to pim. Commun. ACM, 66(6):48–55, 2023.

[5] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions. CoRR, abs/1802.00320, 2018.

[6] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David T. Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. In MICRO, pages 259–272. ACM, 2017.

[7] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In ISCA, pages 14–26. IEEE Computer Society, 2016.

[8] Peyman Pouyan, Esteve Amat, Said Hamdioui, and Antonio Rubio. RRAM variability and its mitigation schemes. In PATMOS, pages 141–146. IEEE, 2016.

[9] Xiao Liu, Mingxuan Zhou, Tajana Simunic Rosing, and Jishen Zhao. HR^{3}AM: A Heat Resilient Design for RRAM-based Neuromorphic Computing. In ISLPED, pages 1–6. IEEE, 2019.

[10] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G. Friedman, Avinoam Kolodny, and Uri C. Weiser. MAGIC - Memristor-Aided Logic. IEEE Trans. Circuits Syst. II Express Briefs, 61-II(11):895–899, 2014.

[11] João Vieira, Edouard Giacomin, Yasir Mahmood Qureshi, Marina Zapater, Xifan Tang, Shahar Kvatinsky, David Atienza, and Pierre-Emmanuel Gaillardon. Accelerating Inference on Binary Neural Networks with Digital RRAM Processing. In VLSI-SoC (Selected Papers), volume 586 of IFIP Advances in Information and Communication Technology, pages 257–278. Springer, 2019.

[12] João Vieira, Nuno Roma, Pedro Tomás, Paolo Ienne, and Gabriel Falcão. Exploiting Compute Caches for Memory Bound Vector Operations. In SBAC-PAD, pages 197–200. IEEE, 2018.

[13] João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. Processing Convolutional Neural Networks on Cache. In International Conference on Acoustics, Speech, and Signal Processing (ICASSP), pages 1658–1662. IEEE, 2020.

[14] João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. A Compute Cache System for Signal Processing Applications. Journal of Signal Processing Systems (JSPS), 93(10):1173–1186, 2021.

[15] Daichi Fujiki, Scott A. Mahlke, and Reetuparna Das. Duality cache for data parallel acceleration. In ISCA, pages 397–410. ACM, 2019.

[16] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. IEEE Comput. Archit. Lett., 15(1):45–49, 2016.

[17] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo F. Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning. In DAC, page 27. ACM, 2019.

[18] Yasir Mahmood Qureshi, William Andrew Simon, Marina Zapater, David Atienza, and Katzalin Olcoz. Gem5-X: A Gem5-Based System Level Simulation Framework to Optimize Many-Core Platforms. In SpringSim, pages 1–12. IEEE, 2019.

[19] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David M. Brooks. Co-designing accelerators and soc interfaces using gem5-aladdin. In MICRO, pages 48:1–48:12. IEEE Computer Society, 2016.

[20] Samuel Rogers, Joshua Slycord, Mohammadreza Baharani, and Hamed Tabkhi. gem5-salam: A system architecture for llvm-based accelerator modeling. In MICRO, pages 471–482. IEEE, 2020.

[21] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. SIGARCH Comput. Archit. News, 39(2):1–7, 2011.

[22] João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. gem5-ndp: Near-Data Processing Architecture Simulation From Low Level Caches to DRAM. In International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD), pages 197–200. IEEE, 2022.

[23] João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. gem5-accel: A pre-rtl simulation toolchain for accelerator architecture validation. IEEE Computer Architecture Letters (CAL), 23(1):1–4, 2024.

[24] João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. Ndpmulator: Enabling full-system simulation for near-data accelerators from caches to DRAM. IEEE Access, 12:10349–10365, 2024.

[25] Hyojong Kim, Ramyad Hadidi, Lifeng Nai, Hyesoon Kim, Nuwan Jayasena, Yasuko Eckert, Onur Kayiran, and Gabriel H. Loh. CODA: enabling co-location of computation and data for near-data processing. CoRR, abs/1710.09517, 2017.

[26] Abhinav Agrawal, Gabriel H. Loh, and James Tuck. Leveraging near data processing for high-performance checkpoint/restart. In SC, page 60. ACM, 2017.

[27] Patrick Siegl, Rainer Buchty, and Mladen Berekovic. Data-centric computing frontiers: A survey on processing-in-memory. In MEMSYS, pages 295–308. ACM, 2016.

[28] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. Attacc! unleashing the power of PIM for batched transformer-based generative model inference. In ASPLOS (2), pages 103–119. ACM, 2024.

[29] Lingxi Wu, Rasool Sharifi, Marzieh Lenjani, Kevin Skadron, and Ashish Venkat. Sieve: Scalable in-situ dram-based accelerator designs for massively parallel k-mer matching. In ISCA, pages 251–264. IEEE, 2021.

[30] Heesu Kim, Hanmin Park, Taehyun Kim, Kwanheum Cho, Eojin Lee, Soojung Ryu, Hyuk-Jae Lee, Kiyoung Choi, and Jinho Lee. Gradpim: A practical processing-in-dram architecture for gradient descent. In HPCA, pages 249–262. IEEE, 2021.

[31] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Dennis Sylvester, David T. Blaauw, Reetuparna Das, and Ravi R. Iyer. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. IEEE Micro, 39(3):11–19, 2019.

[32] Fengbin Tu, Yiqi Wang, Ling Liang, Yufei Ding, Leibo Liu, Shaojun Wei, Shouyi Yin, and Yuan Xie. SDP: co-designing algorithm, dataflow, and architecture for in-sram sparse NN acceleration. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., 42(1):109–121, 2023.

[33] Chen Nie, Chenyu Tang, Jie Lin, Huan Hu, Chenyang Lv, Ting Cao, Weifeng Zhang, Li Jiang, Xiaoyao Liang, Weikang Qian, et al. Vspim: Sram processing-in-memory dnn acceleration via vector-scalar operations. IEEE Transactions on Computers, 2023.

[34] XiaoLiang Hong, Desmond JiaJun Loy, Putu Andhita Dananjaya, Funan Tan, CheeMang Ng, and WenSiang Lew. Oxide-based rram materials for neuromorphic computing. Journal of materials science, 53:8720–8746, 2018.

[35] Shihui Yin, Xiaoyu Sun, Shimeng Yu, and Jae-sun Seo. High-throughput in-memory computing for binary deep neural networks with monolithically integrated RRAM and 90nm CMOS. CoRR, abs/1909.07514, 2019.

[36] Sparsh Mittal. A survey of reram-based architectures for processing-in-memory and neural networks. Mach. Learn. Knowl. Extr., 1(1):75–114, 2019.

[37] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie S. Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology. In MICRO, pages 273–287. ACM, 2017.

[38] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W. Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In ASPLOS, pages 715–731. ACM, 2019.

[39] Bruce L. Jacob, Spencer W. Ng, and David T. Wang. Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann, 2008.

[40] IBM. Understanding dram operation, 1996.

[41] HP. Memory technology evolution: An overview of system memory technologies, 2008.

[42] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In DAC, pages 173:1–173:6. ACM, 2016.

[43] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In ISCA, pages 336–348. ACM, 2015.

[44] Vivek Seshadri, Kevin Hsieh, Amirali Boroumand, Donghyuk Lee, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Fast Bulk Bitwise AND and OR in DRAM. Computer Architecture Letters, 14(2):127–131, 2015.

[45] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri-Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. SIMDRAM: a framework for bit-serial SIMD processing using DRAM. In ASPLOS, pages 329–345. ACM, 2021.

[46] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Rowclone: fast and energy-efficient in-dram bulk data copy and initialization. In MICRO, pages 185–197. ACM, 2013.

[47] João Dinis Ferreira, Gabriel Falcão, Juan Gómez-Luna, Mohammed Alser, Lois Orosa, Mohammad Sadrosadati, Jeremie S. Kim, Geraldo F. Oliveira, Taha Shahroodi, Anant Nori, and Onur Mutlu. pluto: Enabling massively parallel computation in DRAM via lookup tables. In MICRO, pages 900–919. IEEE, 2022.

[48] Kevin K. Chang, Prashant J. Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K. Qureshi, and Onur Mutlu. Low-cost inter-linked subarrays (LISA): enabling fast inter-subarray data movement in DRAM. In HPCA, pages 568–580. IEEE Computer Society, 2016.

[49] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (SALP) in DRAM. In ISCA, pages 368–379. IEEE Computer Society, 2012.

[50] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: a dram-based reconfigurable in-situ accelerator. In MICRO, pages 288–301. ACM, 2017.

[51] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In SC, pages 52:1–52:12. ACM, 2011.

[52] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In MICRO, pages 469–480. ACM, 2009.

[53] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy II, Jack Wadden, Mircea R. Stan, and Kevin Skadron. An overview of micron's automata processor. In CODES+ISSS, pages 14:1–14:3. ACM, 2016.

[54] H.-S. Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T. Chen, and Ming-Jinn Tsai. Metal-oxide RRAM. Proceedings of the IEEE, 100(6):1951–1970, 2012.

[55] Pushen Zuo, Zhong Sun, and Ru Huang. Extremely-fast, energy-efficient massive MIMO precoding with analog RRAM matrix computing. IEEE Trans. Circuits Syst. II Express Briefs, 70(7):2335–2339, 2023.

[56] Jingyu He, Yucong Huang, Miguel Lastras, Terry Tao Ye, Chi-Ying Tsui, and Kwang-Ting Cheng. Rvcomp: Analog variation compensation for rram-based in-memory computing. In ASP-DAC, pages 246–251. ACM, 2023.

[57] Yiwei Du, Jianshi Tang, Yijun Li, Yue Xi, Bin Gao, He Qian, and Huaqiang Wu. Monolithic 3d integration of fefet, hybrid CMOS logic and analog RRAM array for energy-efficient reconfigurable computing-in-memory architecture. In VLSI Technology and Circuits, pages 1–2. IEEE, 2023.

[58] Edouard Giacomin, Tzofnat Greenberg-Toledo, Shahar Kvatinsky, and Pierre-Emmanuel Gaillardon. A robust digital rram-based convolutional block for low-power image processing and learning applications. IEEE Trans. on Circuits and Systems, 66-I(2):643–654, 2019.

[59] João Vieira, Edouard Giacomin, Yasir Mahmood Qureshi, Marina Zapater, Xifan Tang, Shahar Kvatinsky, David Atienza, and Pierre-Emmanuel Gaillardon. A Product Engine for Energy-Efficient Execution of Binary Neural Networks Using Resistive Memories. In VLSI-SoC, pages 160–165. IEEE, 2019.

[60] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In ECCV (4), volume 9908 of Lecture Notes in Computer Science, pages 525–542. Springer, 2016.

[61] André Santos, João Dinis Ferreira, Onur Mutlu, and Gabriel Falcão. Redbit: An end-to-end flexible framework for evaluating the accuracy of quantized cnns. CoRR, abs/2301.06193, 2023.

[62] Ivan Fernandez, Christina Giannoula, Aditya Manglik, Ricardo Quislant, Nika Mansouri-Ghiasi, Juan Gómez-Luna, Eladio Gutiérrez, Oscar G. Plata, and Onur Mutlu. MATSA: an mram-based energy-efficient accelerator for time series analysis. IEEE Access, 12:36727–36742, 2024.

[63] Li Luo, Bochang Li, Lidan Wang, Jinpei Tan, Shukai Duan, and Chunxiang Zhu. Reconfigurable stateful logic circuit with cu/cui/pt memristors for in-memory computing. IEEE Trans. Very Large Scale Integr. Syst., 32(5):835–847, 2024.

[64] Siao-Ping Sing, Ya-Ching Wang, Wei-Hwa Lin, Yue-Der Chih, Yih Wang, Ya-Chin King, and Chrong Jung Lin. A new high density 3d stackable via rram for computing-in-memory soc applications. IEEE Transactions on Electron Devices, 71(4):2399–2403, 2024.

[65] Sven Thijssen, Muhammad Rashedul Haq Rashed, Sumit Kumar Jha, and Rickard Ewetz. Read-based in-memory computing using sentential decision diagrams. In 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC), pages 818–823, 2024.

[66] Chandan Kumar Jha, Sallar Ahmadi-Pour, and Rolf Drechsler. Input distribution aware library of approximate adders based on memristor-aided logic. In VLSID, pages 577–582. IEEE, 2024.

[67] Biagio Peccerillo, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. J. Syst. Archit., 129:102561, 2022.

[68] Yann Falevoz and Julien Legriel. Energy efficiency impact of processing in memory: A comprehensive review of workloads on the UPMEM architecture. In Euro-Par Workshops, volume 14352 of Lecture Notes in Computer Science, pages 155–166. Springer, 2023.

[69] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. CoRR, abs/2105.03814, 2021.

[70] Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, Mohammad Dashti, Romaric Jodin, Alexandre Ghiti, Jordi Chauzi, and Alexandra Fedorova. A case study of processing-in-memory in off-the-shelf systems. In USENIX Annual Technical Conference, pages 117–130. USENIX Association, 2021.

[71] Liang-Chi Chen, Chien-Chung Ho, and Yuan-Hao Chang. Uppipe: A novel pipeline management on in-memory processors for rna-seq quantification. In DAC, pages 1–6. IEEE, 2023.

[72] Safaa Diab, Amir Nassereldine, Mohammed Alser, Juan Gómez-Luna, Onur Mutlu, and Izzat El Hajj. A framework for high-throughput sequence alignment using real processing-in-memory systems. Bioinform., 39(5), 2023.

[73] Dominique Lavenier, Remy Cimadomo, and Romaric Jodin. Variant calling parallelization on processor-in-memory architecture. In BIBM, pages 204–207. IEEE, 2020.

[74] Arthur Bernhardt, Andreas Koch, and Ilia Petrov. pimdb: From main-memory DBMS to processing-in-memory dbms-engines on intelligent memories. In DaMoN, pages 44–52. ACM, 2023.

[75] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim. Design and analysis of a processing-in-dimm join algorithm: A case study with UPMEM dimms. Proc. ACM Manag. Data, 1(2):113:1–113:27, 2023.

[76] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. Adaptive query compilation with processing-in-memory. In ICDEW, pages 191–197. IEEE, 2023.

[77] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. Accelerating large table scan using processing-in-memory technology. Datenbank-Spektrum, 23(3):199–209, 2023.

[78] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. Pim-tree: A skew-resistant index for processing-in-memory. Proc. VLDB Endow., 16(4):946–958, 2022.

[79] Juan Gómez-Luna, Yuxin Guo, Sylvan Brocard, Julien Legriel, Remy Cimadomo, Geraldo F. Oliveira, Gagandeep Singh, and Onur Mutlu. An experimental evaluation of machine learning training on a real processing-in-memory system. CoRR, abs/2207.07886, 2022.

[80] Prangon Das, Purab Ranjan Sutradhar, Mark A. Indovina, Sai Manoj Pudukotai Dinakarrao, and Amlan Ganguly. Implementation and evaluation of deep neural networks in commercially available processing in memory hardware. In SOCC, pages 1–6. IEEE, 2022.

[81] Harshita Gupta, Mayank Kabra, Juan Gómez-Luna, Konstantinos Kanellopoulos, and Onur Mutlu. Evaluating homomorphic operations on a real-world processing-in-memory system. In IISWC, pages 211–215. IEEE, 2023.

[82] Joonyoung Kim and Younsu Kim. HBM: memory solution for bandwidth-hungry processors. In Hot Chips Symposium, pages 1–24. IEEE, 2014.

[83] Hongshin Jun, Sang Kyun Nam, Han Ho Jin, Jong-Chern Lee, Yong Jae Park, and Jaejin Lee. High-bandwidth memory (HBM) test challenges and solutions. IEEE Des. Test, 34(1):16–25, 2017.

[84] Gabriel H. Loh. 3d-stacked memory architectures for multi-core processors. In ISCA, pages 453–464. IEEE Computer Society, 2008.

[85] Daehyun Kim, Krit Athikulwongse, Michael B. Healy, Mohammad M. Hossain, Moongon Jung, Ilya Khorosh, Gokul Kumar, Young-Joon Lee, Dean L. Lewis, Tzu-Wei Lin, Chang Liu, Shreepad Panth, Mohit Pathak, Minzhen Ren, Guanhao Shen, Taigon Song, Dong Hyuk Woo, Xin Zhao, Joungho Kim, Ho Choi, Gabriel H. Loh, Hsien-Hsin S. Lee, and Sung Kyu Lim. Design and analysis of 3d-maps (3D massively parallel processor with stacked memory). IEEE Trans. Computers, 64(1):112–125, 2015.

[86] Dylan C. Stow, Itir Akgun, Wenqin Huangfu, Yuan Xie, Xueqi Li, and Gabriel H. Loh. Efficient system architecture in the era of monolithic 3d: Dynamic inter-tier interconnect and processing-in-memory. In DAC, page 100. ACM, 2019.

[87] J. Thomas Pawlowski. Hybrid memory cube (HMC). In Hot Chips Symposium, pages 1–24. IEEE, 2011.

[88] Dong-Uk Lee, Kyung Whan Kim, Kwan-Weon Kim, Kang Seol Lee, Sang Jin Byeon, Jin-Hee Cho, Han Ho Jin, Sang Kyun Nam, Jaejin Lee, Jun Hyun Chun, and Sung-Joo Hong. An exact measurement and repair circuit of TSV connections for 128gb/s high-bandwidth memory(hbm) stacked DRAM. In VLSIC, pages 1–2. IEEE, 2014.

[89] Jin Hyun Kim, Shinhaeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, BengSeng Phuah, Jihyun Choi, Jinin So, YeonGon Cho, Joon-Ho Song, Jangseok Choi, Jeonghyeon Cho, Kyomin Sohn, Young-Soo Sohn, Kwang-Il Park, and Nam Sung Kim. Aquabolt-xl: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond. In HCS, pages 1–26. IEEE, 2021.

[90] Jin Hyun Kim, Shinhaeng Kang, Sukhan Lee, Hyeonsu Kim, Yuhwan Ro, Seungwon Lee, David Wang, Jihyun Choi, Jinin So, YeonGon Cho, Joon-Ho Song, Jeonghyeon Cho, Kyomin Sohn, and Nam Sung Kim. Aquabolt-xl hbm2-pim, LPDDR5-PIM with in-memory processing, and AXDIMM with acceleration buffer. IEEE Micro, 42(3):20–30, 2022.

[91] Shinhaeng Kang, Sukhan Lee, Byeongho Kim, Hweesoo Kim, Kyomin Sohn, Nam Sung Kim, and Eojin Lee. An fpga-based RNN-T inference accelerator with PIM-HBM. In FPGA, pages 146–152. ACM, 2022.

[92] Donghun Lee, Jinin So, Minseon Ahn, Jong-Geon Lee, Jungmin Kim, Jeonghyeon Cho, Oliver Rebholz, Vishnu Charan Thummala, Ravi Shankar JV, Sachin Suresh Upadhya, Mohammed Ibrahim Khan, and Jin Hyun Kim. Improving in-memory database operations with acceleration DIMM (axdimm). In DaMoN, pages 2:1–2:9. ACM, 2022.

[93] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sánchez, and Nathan Beckmann. Livia: Data-Centric Computing Throughout the Memory Hierarchy. In ASPLOS, pages 417–433. ACM, 2020.

[94] Sergiu Mosanu, Mohammad Nazmus Sakib, Tommy Tracy II, Ersin Cukurtas, Alif Ahmed, Preslav Ivanov, Samira Manabi Khan, Kevin Skadron, and Mircea Stan. Pimulator: a fast and flexible processing-in-memory emulation platform. In DATE, pages 1473–1478. IEEE, 2022.

[95] Florent Kermarrec, Sébastien Bourdeauducq, Jean-Christophe Le Lann, and Hannah Badier. Litex: an open-source soc builder and library based on migen python DSL. CoRR, abs/2005.02506, 2020.

[96] Steven J. E. Wilton and Norman P. Jouppi. CACTI: an enhanced cache access and cycle time model. IEEE J. Solid State Circuits, 31(5):677–688, 1996.

[97] Daniel Sánchez and Christos Kozyrakis. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In ISCA, pages 475–486. ACM, 2013.

[98] William Andrew Simon, Yasir Mahmood Qureshi, Alexandre Levisse, Marina Zapater, and David Atienza. BLADE: A BitLine Accelerator for Devices on the Edge. In ACM Great Lakes Symposium on VLSI, pages 207–212. ACM, 2019.

[99] Chao Yu, Sihang Liu, and Samira Manabi Khan. MultiPIM: A Detailed and Configurable Multi-Stack Processing-In-Memory Simulator. IEEE Comput. Archit. Lett., 20(1):54–57, 2021.

[100] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In HPCA, pages 7–18. IEEE Computer Society, 2003.

[101] Marius Herget, Faezeh Sadat Saadatmand, Martin Bor, Ignacio González Alonso, Todor P. Stefanov, Benny Akesson, and Andy D. Pimentel. Design space exploration for distributed cyber-physical systems: State-of-the-art, challenges, and directions. In DSD, pages 632–640. IEEE, 2022.

[102] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. To PIM or not for emerging general purpose processing in DDR memory systems. In ISCA, pages 231–244. ACM, 2022.

[103] Giovanni Mariani, Gianluca Palermo, Vittorio Zaccaria, and Cristina Silvano. OSCAR: an optimization methodology exploiting spatial correlation in multicore design spaces. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., 31(5):740–753, 2012.

[104] Giovanni Mariani, Gianluca Palermo, Vittorio Zaccaria, and Cristina Silvano. Design-space exploration and runtime resource management for multicores. ACM Trans. Embed. Comput. Syst., 13(2):20:1–20:27, 2013.

[105] Rik Jongerius, Andreea Anghel, Gero Dittmann, Giovanni Mariani, Erik Vermij, and Henk Corporaal. Analytic multi-core processor model for fast design-space exploration. IEEE Trans. Computers, 67(6):755–770, 2018.

[106] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Linux device drivers - where the Kernel meets the hardware (3. ed.). O'Reilly, 2005.

[107] Ettore Tiotto, Victor Perez, Whitney Tsang, Lukas Sommer, Julian Oppermann, Victor Lomüller, Mehdi Goli, and James Brodman. Experiences building an mlir-based SYCL compiler. In CGO, pages 399–410. IEEE, 2024.

[108] Horácio C. Neto and Mário P. Véstias. Very low resource table-based FPGA evaluation of elementary functions. In ReConFig, pages 1–6. IEEE, 2013.

[109] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In IISWC, pages 44–54. IEEE Computer Society, 2009.

[110] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In CVPR, pages 779–788. IEEE Computer Society, 2016.

[111] Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement. CoRR, abs/1804.02767, 2018.

[112] Palash Das and Hemangee K. Kapoor. Towards near-data processing of compare operations in 3d-stacked memory. In ACM Great Lakes Symposium on VLSI, pages 243–248. ACM, 2018.

[113] Zhengrong Wang, Christopher Liu, and Tony Nowatzki. Infinity stream: Enabling transparent and automated in-memory computing. IEEE Comput. Archit. Lett., 21(2):85–88, 2022.

[114] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. Near-stream computing: General and transparent near-cache acceleration. In HPCA, pages 331–345. IEEE, 2022.

[115] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert J. Ou, Colin Schmidt, Samuel Steffl, John Charles Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In DAC, pages 769–774. IEEE, 2021.

[116] Joseph Redmon. Darknet: Open Source Neural Networks in C. `http://pjreddie.com/darknet/`, 2013–2016.

[117] Reese Kuper, Ipoom Jeong, Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Jiayu Hu, Sanjay Kumar, Philip Lantz, and Nam Sung Kim. A quantitative analysis and guide-

lines of data streaming accelerator in modern intel xeon scalable processors. In ASPLOS (2), pages 37–54. ACM, 2024.

[118] Ioannis Stamoulias, Christoforos Kachris, and Dimitrios Soudris. Hardware accelerators for financial applications in HDL and high level synthesis. In SAMOS, pages 278–285. IEEE, 2017.

[119] Zhewen Yu and Christos-Savvas Bouganis. Autows: Automate weights streaming in layer-wise pipelined DNN accelerators. CoRR, abs/2311.04764, 2023.

[120] Jingwei Cai, Zuotong Wu, Sen Peng, Yuchen Wei, Zhanhong Tan, Guiming Shi, Mingyu Gao, and Kaisheng Ma. Gemini: Mapping and architecture co-exploration for large-scale DNN chiplet accelerators. In HPCA, pages 156–171. IEEE, 2024.

[121] Cristina Silvano, Daniele Ielmini, Fabrizio Ferrandi, Leandro Fiorin, Serena Curzel, Luca Benini, Francesco Conti, Angelo Garofalo, Cristian Zambelli, Enrico Calore, Sebastiano Fabio Schifano, Maurizio Palesi, Giuseppe Ascia, Davide Patti, Stefania Perri, Nicola Petra, Davide De Caro, Luciano Lavagno, Teodoro Urso, Valeria Cardellini, Gian Carlo Cardarilli, and Robert Birke. A survey on deep learning hardware accelerators for heterogeneous HPC platforms. CoRR, abs/2306.15552, 2023.

[122] Hanqiu Chen and Cong Hao. Hardware/software co-design for machine learning accelerators. In FCCM, pages 233–235. IEEE, 2023.

[123] Ghattas Akkad, Ali Mansour, and Elie Inaty. Embedded deep learning accelerators: A survey on recent advances. IEEE Trans. Artif. Intell., 5(5):1954–1972, 2024.

[124] Sungyeob Yoo, Hyunsung Kim, Jinseok Kim, Sunghyun Park, Joo-Young Kim, and Jinwook Oh. Lighttrader: A standalone high-frequency trading system with deep learning inference accelerators and proactive scheduler. In HPCA, pages 1017–1030. IEEE, 2023.

[125] Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. Unlimited vector extension with data streaming support. In ISCA, pages 209–222. IEEE, 2021.

[126] Nuno Neves, Joao Mario Domingos, Nuno Roma, Pedro Tomás, and Gabriel Falcão. Compiling for vector extensions with stream-based specialization. IEEE Micro, 42(5):49–58, 2022.

Publications

Publications in International Scientific Journals

- João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. Processing Convolutional Neural Networks on Cache. In International Conference on Acoustics, Speech, and Signal Processing (ICASSP), pages 1658–1662. IEEE, 2020
- João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. gem5-ndp: Near-Data Processing Architecture Simulation From Low Level Caches to DRAM. In International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD), pages 197–200. IEEE, 2022

Publications in International Scientific Conferences

- João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. A Compute Cache System for Signal Processing Applications. Journal of Signal Processing Systems (JSPS), 93(10):1173–1186, 2021
- João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. gem5-accel: A pre-rtl simulation toolchain for accelerator architecture validation. IEEE Computer Architecture Letters (CAL), 23(1):1–4, 2024
- João Vieira, Nuno Roma, Gabriel Falcão, and Pedro Tomás. Ndpmulator: Enabling full-system simulation for near-data accelerators from caches to DRAM. IEEE Access, 12:10349–10365, 2024