

Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electrical, Electronics and Communications Engineering (36^{th}cycle)

From Control to Compute: Modular CPU design and Programmable Near-Memory Solutions for Edge Applications

By

Michele CAON

******

Supervisor(s):
Prof. Maurizio MARTINA, Supervisor
Prof. Guido MASERA, Co-Supervisor

Doctoral Examination Committee:
Prof. Sergio SAPONARA, University of Pisa
Prof. Mario KOVA, University of Zagreb
Prof. Luciano LAVAGNO, Politecnico di Torino
Dr. Pasquale Davide Schiavone, EPFL
Dr. Maurizio Capra, Synthara AG

Politecnico di Torino
2024

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Michele CAON
2024

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

To curiosity, to mistakes.

Acknowledgements

Okay, so let's get real. It's 7:15 AM. I'm famished, fatigued, and fashionably late. So I'll keep this short and sweet. Bittersweet, maybe.

Let's start with a huge shoutout to the real MVPs of this journey: my comrades. The ones who didn't just put up with my existential dread or half-baked ideas but joined me in one or the other. You were the real deal, and I'm grateful for all the kebabs, Negronis, coffee-flavoured therapy time, and camper-shaped clothes dryers we shared. Plus, I wouldn't be writing any of this if it weren't for you–which is a good thing, I guess.

Thank you, friends, for still being friends after repeated weeks of sudden radio silence.

Mom and Dad. I know you'll read the next hundred pages despite not understanding half of them, and I love it. You instilled the curiosity that led me to this point. Però avete fatto anche un sacco di cose belle. Grazie.

Thank you, procrastination, for always being there when I needed you most. Thank you, mighty AI, for helping me meet deadlines despite the previous sentence. Sort of. And thank you to my supervisor for patiently dealing with the last two sentences and for letting me explore every single rabbit hole I found along the way. I sincerely value that freedom in this academic wasteland of metrics over meaning. Yet I also appreciate those colleagues who pulled me out of the rabbit hole, ensuring I had something to write in this thesis.

Finally, thank you, past me, for making random decisions that somehow always end up with present me insulting you. It's always a pleasure.

Thank you, Ossidiana.

Cheers.

Abstract

The widespread adoption of data-centric algorithms, particularly Artificial Intelligence (AI) and Machine Learning (ML), has exposed the limitations of centralized processing infrastructures, driving a shift towards edge computing. This shift necessitates stringent constraints on energy efficiency, which traditional von Neumann architectures struggle to meet. The Compute-In-Memory (CIM) paradigm has emerged as a superior candidate due to its efficient exploitation of available memory bandwidth. However, existing CIM solutions require high implementation effort and lack flexibility from a software integration standpoint.

In this thesis, the von Neumann bottleneck is tackled by a combination of CIM and Instruction-Level Parallelism (ILP) exploitation, resulting in an energy-efficient and fully programmable approach that enables the development of the next generation of intelligent edge computing nodes.

The proposed NM-Carus Near-Memory Computing (NMC) module serves as a programmable memory architecture replacing conventional SRAM banks, to which data-intensive computation kernels can be offloaded. When implemented on a low-power 65 nm technology, NM-Carus achieves a peak energy efficiency of 306.7 GOPS/W, outperforming existing state-of-the-art in- and near-memory solutions while providing a flexible Instruction Set Architecture (ISA). On the other hand, the novel LEN5 RISC-V CPU implements a modular, scalable, and easily extensible microarchitecture that exploits ILP to maximize the Instructions Per Cycle (IPC) when executing long-latency instructions such as those implemented by NM-Carus. LEN5 significantly surpasses the performance of existing embedded-class CPUs with a moderate 10 % area overhead when embedded in an ultra-low-power Microcontroller Unit (MCU) with 256 KiB of SRAM memory.

The development and consolidation of these two architectures represent a solid foundation for the design of energy-efficient, memory-centric computing nodes, while enabling seamless integration of application software through the use of existing programming languages and deployment tools.

As a future research direction, the integration of LEN5 and NM-Carus into a unified systemsuch as the proposed ARCANE architecturecan further enhance performance and energy efficiency. ARCANE aims to combine tightly coupled in-cache computing with out-of-order execution capabilities, leveraging the strengths of both NM-Carus and LEN5 to address the limitations of loosely coupled near-memory computing and fully exploit instruction-level parallelism in edge computing applications.

Contents

List of Figures **x**

List of Tables **xiii**

List of Listings **xiv**

Acronyms and Abbreviations **xv**

1 Introduction **1**

1.1 Historical Context 1

1.2 Motivation 2

1.3 Contributions and Publications 5

1.3.1 Programmable Near-Memory Computing Architecture 7

1.3.2 Modular, Dynamically Scheduled RISC-V CPU Core 8

1.3.3 Approximate ANN Inference on Embedded Microcontrollers 10

1.4 Thesis Outline 11

2 NM-Carus: Programmable Near-Memory Computing **12**

2.1 Limitations of Current Compute-In-Memory Architectures 12

2.1.1 In-Memory Computing Limitations 14

2.1.2 Near-Memory Computing as a Practical Solution 14

2.1.3 The Need for Programmable, SRAM-Based CIM Solutions 15

2.2 Proposed Near-Memory Computing Architecture 16
2.2.1 Design Philosophy 16
2.2.2 Instruction Set Architecture 21
2.2.3 Microarchitecture 27
2.2.4 Microarchitecture 27
2.2.5 Architectural Comparison from a Software Perspective 33
2.3 NM-Carus Physical Implementation 36
2.3.1 Design Constraints and Methodology 36
2.3.2 Post-Layout Characteristics 37
2.3.3 Implementation Details of NM-Carus 38
2.4 NM-Carus Performance Assessment 39
2.4.1 Experimental Setup 40
2.4.2 Experimental Results 41
2.4.3 Comparison with the State of the Art 43

3 LEN5: Out-of-Order CPU Design for Edge Computing 51
3.1 Limitations of In-Order CPUs in Edge Computing 51
3.2 Proposed Out-of-Order Microarchitecture 53
3.2.1 Instruction Fetch and Branch Prediction 54
3.2.2 Out-of-Order Instruction Execution 55
3.2.3 Load and Store Handling 56
3.2.4 Out-of-Order Instruction Commit 56
3.2.5 Latency Hiding and Dependency Resolution 57
3.2.6 Scalability and Modularity 58
3.2.7 Implementation Considerations 58
3.3 LEN5 Physical Implementation 59
3.4 Performance Assessment 61
3.4.1 Synthetic Benchmarking with Configurable Coprocessor 61

3.4.2 Instruction Per Cycle (IPC) Analysis 62
3.4.3 Real-World Benchmarking 66
3.4.4 Discussion 67
3.4.5 Conclusion 67

4 Conclusion and Future Work 68
4.1 Summary of Contributions 68
4.1.1 NM-Carus: A Programmable Near-Memory Computing Architecture 68
4.1.2 LEN5: An Out-of-Order Central Processing Unit (CPU) Optimized for Coprocessor Utilization 69
4.2 Limitations and Future Work 70
4.2.1 Limitations of Loosely Coupled Near-Memory Computing 70
4.2.2 Tightly Coupled In-Cache Computing 72
4.2.3 New ILP Opportunities 76
4.2.4 Future Integration and Development 76

References 77

Appendix A Code Snippets 82
A.1 Matrix Multiplication Kernel 82

List of Figures

1.1 MARLIN framework when integrated with the PULP SoC. Source: [25]. . . 10

2.1 Architectural differences between In-Memory Computing (IMC) and Near-Memory Computing (NMC) solutions. In IMC devices, the processing units are embedded within the memory arrays, so accessing the stored data does not require traversing the array periphery. In NMC systems, the processing units are placed near the memory arrays, yet outside the periphery circuitry. 13

2.2 Top-level block diagram of an NMC-enhanced MCU incorporating the proposed NM-Carus as part of its memory subsystem. This setup contrasts with traditional methods that rely on streaming commands over the bus interface. Source: [22]. 17

2.3 Top-level block diagram of NM-Carus. Source: [22]. 29

2.4 Parallel execution of scalar and vector instructions in NM-Carus. Source [22]. 29

2.5 Organization of the NM-Carus Vector Register File (VRF). Source: [22]. . . 30

2.6 Post-synthesis area breakdown of a 32 KiB instance NM-Carus to scale. Data source: [22]. 37

2.7 Layout of the 32 KiB NM-Carus implementation (4 lanes). Source: [22]. . . . 38

2.8 Layout of the X-HEEP system enhanced with NM-Carus. 40

2.9 Energy efficiency gain a and throughput improvement b of the NMC-enhanced X-HEEP Microcontroller Unit (MCU) compared with its CPU-only version. Data source: post-layout simulation (65 nm, $f_{\mathrm{CLK}} = 250\,\mathrm{MHz}$). 46

2.10 Throughput a and energy b scaling of NM-Carus (cyan) compared to the CPU-only MCU version (grey) on several matrix multiplication kernels with different shapes and bitwidths. The CPU throughput does not vary significantly with data size when using the standard RV32IMC Instruction Set Architecture (ISA); therefore, only 32-bit data is shown. Data source: post-layout simulation. 47

2.11 Average power breakdown in CPU (grey) and NM-Carus (cyan) for 8-bit and 32-bit 2D convolution a and matrix multiplication b. Data source: post-layout simulation. 48

2.12 Comparison of the normalized performance and characteristics of NM-Carus with state-of-the-art architectures. *Energy efficiency*, *Throughput*, and *Useful bitcell density* are quantitative metrics taken from Tables 2.5 and 2.6. The *Versatility* score considers qualitative aspects such as ease of use (e.g., programmability in the case of NM-Carus), the range of supported operations (including special-purpose ones like shuffle operations in C-SRAM [52]), and data placement or replication constraints. *Deployment effort* refers to the complexity of the ASIC deployment process and the effort required to port the device to a different technology node. Fully digital NMC architectures using commercial off-the-shelf Static Random-Access Memory (SRAM) are therefore assigned a higher score. 50

3.1 Block diagram of the LEN5 microprocessor. Source: [24]. 54

3.2 Backend architecture of LEN5 with an example of out-of-order execution and commit. The `sw` instruction is waiting for the result (`t2`) produced by the long-latency `coproc` instruction (RAW hazard). Independent instructions (`addi`, `bne`) have completed execution out of order and are eligible for commit. Source: [23]. 57

3.3 Area breakdown of the LEN5 **Max Performance** configuration. 60

3.4 Instructions Per Cycle (IPC) comparison without housekeeping and without dependencies using iterative (left) and pipelined (right) Configurabe-Latency Coprocessor (CLC). Source: [23]. 63

3.5 IPC comparison with housekeeping and without dependencies using iterative (left) and pipelined (right) CLC. Source: [23]. 64

3.6 IPC comparison with dependencies using iterative CLC without housekeeping (left) and with housekeeping (right). Source: [23]. 64

3.7 IPC comparison with dependencies using pipelined CLC without housekeeping (left) and with housekeeping (right). Source: [23]. 65

3.8 IPC comparison as the coprocessor latency and Number of Instructions per Block (NIB) increase, highlighting the impact of ROB size limitations. Source: [23]. 66

3.9 IPC comparison with CV32E40P over the Embench suite (colored bars) and executed instruction composition (grayscale). Source: [24]. 66

4.1 Top-level block diagram of the Adaptive RISC-V Cache Architecture for Near-Memory Extensions (ARCANE) system with detailed architecture of its NM-Carus-based data cache. 72

List of Tables

2.1 `xvnmc` instruction listing. Instruction variants are expanded in Table 2.2. Vector arithmetic operations are element-wise unless otherwise specified. Instruction with the `[r]` type can optionally use indirect register addressing. 23

2.2 NM-Carus instruction formats. The `xvnmc` extension is implemented inside the RISC-V *Custom-2* 25-bit encoding space with the `0x5b` major opcode. The standard OPIVV, OPIVX, and OPIVI formats are used for the `vv`, `vx`, and `vi` instruction variants respectively, while `ex` and `xe` use the OPMVX format. Indirect register addressing is available for all `vv`, `vx`, and `vi` instructions. . . 24

2.3 Post-layout area and timing characteristics of a 32 KiB traditional SRAM and NM-Carus when implemented on a low-power 65 nm CMOS technology node. 37

2.4 System-level throughput and energy improvement of the X-HEEP system when executing the same kernels on NM-Carus, relative to their CPU-only implementation (baseline). Improvements are computed as CPU data divided by NM-Carus data (higher is better). Data source: post-layout simulation (65 nm, $f_{\mathrm{CLK}} = 250\,\mathrm{MHz}$). 45

2.5 Comparison with existing state-of-the-art IMC and NMC solutions. 49

2.6 Peak performance comparison on matrix multiplication kernels. 50

3.1 Post-synthesis area and clock frequency comparison of LEN5 configurations, CVA6, and CV32E40P when implemented in a low-power 65 nm CMOS technology. 60

List of Listings

1 Example of a NM-Carus kernel for matrix multiplication ($R = A \times B$) using `xvnmc` instructions. Custom `xvnmc` instructions are highlighted in bold. . . . 27
2 Example of an RISC-V Vector Extension (RVV) kernel for matrix multiplication ($R = A \times B$). RVV instructions are highlighted in bold. 28
3 Assembly code for matrix multiplication on a standard Reduced Instruction Set Computer (RISC) CPU. 83

Acronyms and Abbreviations

AI Artificial Intelligence
ALU Arithmetic Logic Unit
ANN Artificial Neural Network
ARCANE Adaptive RISC-V Cache Architecture for Near-Memory Extensions
ASIC Application-Specific Integrated Circuit
AT Address Table
BPU Branch Prediction Unit
BTB Branch Target Buffer
CDB Common Data Bus
CDN Content Delivery Network
CIM Compute-In-Memory
CISC Complex Instruction Set Computer
CLC Configurabe-Latency Coprocessor
CMOS Complementary Metal-Oxide-Semiconductor
CPU Central Processing Unit
CSR Control and Status Register
DLP Data-Level Parallelism
DMA Direct Memory Access
DNN Deep Neural Network
DPU Dot-Product Unit
DSP Digital Signal Processor
eCPU embedded CPU
eMEM embedded Memory
EU Execution Unit
FSM Finite State Machine
GCC GNU Compiler Collection
GEMM General Matrix Multiplication

GPR General-Purpose Register
GPU Graphics Processing Unit
IC Integrated Circuit
ILP Instruction-Level Parallelism
IMC In-Memory Computing
IoT Internet of Things
IPC Instructions Per Cycle
ISA Instruction Set Architecture
ISR Interrupt Service Routine
IQ Issue Queue
LB Load Buffer
LLC Last-Level Cache
LSU Load-Store Unit
OoO Out-of-Order
OS Operating System
MAC Multiply-and-Accumulate
MCU Microcontroller Unit
MRAM Magnetoresistive Random Access Memory
NIB Number of Instructions per Block
NMC Near-Memory Computing
NPU Neural Processing Unit
NVM Non-Volatile Memory
PC Program Counter
PCM Phase-Change Memory
RAS Return Address Stack
RAW Read-After-Write
RF Register File
RISC Reduced Instruction Set Computer
ROB ReOrder Buffer
RRAM Resistive Random Access Memory
RS Reservation Station
RTL Register Transfer Level
RVV RISC-V Vector Extension
SB Store Buffer
SEW Selected Element Width
SIMD Single Instruction Multiple Data

SoC System-on-Chip
SRAM Static Random-Access Memory
VCD Value Change Dump
VPU Vector Processing Unit
VRF Vector Register File
WAR Write-After-Read
WAW Write-After-Write

Chapter 1

Introduction

1.1 Historical Context

Since the introduction of the first operational computers in the 1940s, computing has oscillated between a centralized and a decentralized model, driven by a combination of technological advancements and, more recently, the evolution of social trends and the consequent user needs. In the early days, centralization was a necessity dictated by the sheer complexity, size and cost of the first electromechanical computers [1]. These machines, such as the EDSAC built at the University of Cambridge in 1949 [2], required dedicated facilities and a team of operators to function.

Advancements in computer architecture and the advent of high-level programming languages during the 1960s [3] led to improved automatization in program development and execution, which increased the overall accessibility of computers. This, together with the advances in semiconductor technology and the introduction of the first integrated microprocessors in the 1970s [4], set the stage for a shift towards decentralization and the proliferation of personal computers. By the end of 1980s, individuals and business entities had access to local, affordable computing resources, reducing dependency on centralized mainframes. Consequently, the 1990s saw the rise of the internet, which further enabled distributed computing and peer-to-peer communication. The introduction of concepts like Content Delivery Networks (CDNs) in this period [5] can be seen as a precursor to what would later become edge computing.

However, as the volume of internet traffic exploded in the 2000s and 2010s with the rise of social media, centralized infrastructures re-emerged in the form of *cloud computing* as a

cost-effective solution to the growing demand for storage, processing power, and scalability [6]. The cloud model, which relies on a network of data centers to provide on-demand access to shared resources, has become the backbone of modern computing, enabling the deployment of large-scale applications and services through the internet.

More recently, the increased demand for real-time data processing and the ever-growing diffusion of Artificial Neural Network (ANN) in several aspects of modern life, from digital assistants to autonomous vehicles, has exposed the limitation of the available network infrastructure in terms of latency and bandwidth. With the proliferation of data sources like smartphones, Internet of Things (IoT) devices, and smart sensors, there is now a critical need to perform computation closer to where the data is generated, reducing the reliance on distant cloud servers, enabling reduced latency, enhanced privacy, and limited bandwidth requirements. This trend has made decentralization more relevant than ever, leading to the emergence of *edge computing* [7], [8].

As edge computing continues to evolve, the demand for highly efficient, specialized processing solutions has intensified. While cloud computing remains critical for large-scale data processing, the need for real-time, low-latency computation has pushed the boundaries of the existing network infrastructure and traditional processing platforms. General-purpose microprocessors, despite their flexibility, struggle to meet the performance and energy efficiency requirements of modern, energy-constrained, edge applications, especially in the face of increasing data volume and complexity. Emerging workloads, such as ANN, which heavily rely on data-intensive operations, are placing unprecedented strain on both computational and memory resources [9]. To address these challenges, domain-specific accelerators like Digital Signal Processors (DSPs), Graphics Processing Units (GPUs), and Neural Processing Units (NPUs) were proposed to optimize the use of the available computational resources and memory bandwidth, yet often sacrificing flexibility, complicating the application deployment process. This thesis seeks to address these limitations by exploring innovative approaches to hardware design for edge computing, focusing on enhancing both computational throughput and energy efficiency by combining the energy efficiency brought by the Compute-In-Memory (CIM) paradigm with the flexibility of traditional programmable architectures.

1.2 Motivation

In the context highlighted in the previous section, emerging workloads, particularly in the realm of Artificial Intelligence (AI) in fields like biosignal, audio, and video processing [10]–[12], place increasing demands on computational throughput, that edge devices must provide

while meeting the strict energy constraints imposed by their battery-powered nature. To address these challenges, it has become crucial to develop innovative embedded circuits that can efficiently handle the latency limitations of centralized systems while providing sufficient, if not comparable, processing power to meet the requirements of modern applications.

A key contributor to energy consumption in traditional computing systems, especially when dealing with data-intensive workloads, is memory access. Numerous studies have shown that memory accesses account for a significant portion of the the overall energy usage in modern microprocessor-based computing systems [13], [14], outpacing the energy cost of arithmetic operations. At the root of this inefficiency is the *von Neumann bottleneck*, a fundamental limitation of the conventional von Neumann computer architecture, where the processing elements are physically separated from the memory hierarchy [15], [16]. The resulting constant data movement leads to energy overheads, latency penalties, and bandwidth limitations, which are particularly evident in edge-oriented systems, where the communication between the processing elements and the memory hierarchy is often bottlenecked by the limited bandwidth and concurrency of their simple and energy-efficient bus interconnects. The problem is further exacerbated by the stagnation of SRAM technology, which has failed to keep pace with the scaling trends of integrated logic gates, leading to a widening gap between the processing and memory speeds and additional integration challenges [14].

To address these challenges, the CIM paradigm, which includes IMC and NMC approaches, has emerged as a promising solution. By moving the processing elements within or close to the memory arrays, these architectures aim to reduce the energy and latency penalties associated with data movement and to optimize the increased memory bandwidth utilization by parallelizing data accesses and processing [17]–[19]. However, while true IMC devices, exploiting custom-made memory arrays embedding processing elements, could mostly eliminate the need for data movement, several technological obstacles currently hinder their manufacturability and scalability. While SRAM-based CIM solutions can be manufactured using existing fabrication processes, their scalability and efficiency are limited by the degraded readout precision when embedding processing elements operating on multiple wordlines, with further challenges arising from the increased area overhead, analog circuitry noise, and timing constraints [20]. On the other hand, solutions based on magnetoelectric technologies like Magnetoresistive Random Access Memory (MRAM) and Resistive Random Access Memory (RRAM) still face challenges from a manufacturing perspective, with limited scalability and difficult integration with existing Complementary Metal-Oxide-Semiconductor (CMOS) processes [21]. Therefore, the NMC approach is gaining traction as a more practical solution which leverages the existing standard cell digital design and implementation flow to embed processing elements just outside the periphery of commercially available memory ar-

rays. While this approach is allegedly not as promising as IMC in terms of energy efficiency, it offers a readily manufacturable solution that still mitigates the von Neumann bottleneck and provides a significant improvement in terms of energy efficiency and latency compared to traditional microprocessor-based systems.

In addition to improving energy efficiency, modern edge computing systems must also offer a high degree of flexibility. As workloads continue to evolve and diversify, edge devices must be capable of executing a wide variety of tasks, ranging from control-driven housekeeping, coordination, and data acquisition and transmission functions to computationally-intensive AI inference. While specialized accelerators can significantly enhance performance and energy efficiency for specific tasks, the fully programmable and general-purpose nature of CPUs remain central to the versatility and adaptability of modern computing systems. Therefore, modern edge computing systems are increasingly heterogeneous, comprising domain-specific subsystems alongside general-purpose CPUs to balance the trade-offs between performance, energy efficiency, and flexibility. In this context, CIM architectures can provide a viable alternative to conventional accelerators, offering reduced data movement and increased memory bandwidth utilization. However, existing solutions suffer from reduced flexibility and poor integration with the host system, limiting their applicability to a narrow range of workloads. The work presented in Chapter 2 aims to address these limitations by proposing a novel approach to NMC, called NM-Carus, that allows the memory subsystem to be programmed with a familiar Single Instruction Multiple Data (SIMD) ISA and configured dynamically by the host CPU to execute arbitrary workloads, bridging the gap between the energy efficiency of CIM and the flexibility of traditional microprocessor at a lower area and power overhead compared to conventional accelerators.

Nevertheless, the offloading of complex operations from the CPU to domain-specific subsystems introduces new challenges. These tasks often have long execution latencies, especially when they involve data-intensive operations or entire small programs. To maximize overall system performance, the system CPU must be optimized to handle control operations, like housekeeping tasks, in parallel to these long-latency instructions. However, conventional edge-oriented embedded systems rely on simple in-order CPUs that are not well-suited to exploit the available Instruction-Level Parallelism (ILP) in the program. In this context, techniques to improve the IPC can be borrowed from high-performance, application class microprocessors to improve the overall system throughput. In particular, by dynamically scheduling instructions based on their data dependencies and control flow, Out-of-Order (OoO) CPUs can extract a high degree of parallelism from the instruction stream. Branch prediction and speculative instruction execution can further enhance the IPC by allowing the CPU to consider a wider range of instructions for execution, even in the presence of control hazards, ensuring a

higher utilization of the available computing resources within and outside the CPU core. The work presented in Chapter 3 aims at bringing these techniques to the edge computing domain by proposing a novel RISC-V CPU core that combines a modular, latency-independent and scalable execution pipeline with a dynamically scheduled, OoO execution model. The resulting core, called LEN5, is designed to provide higher IPC compared to state-of-the-art in-order CPUs while maintaining a low system-level area overhead.

Thus, the work presented in this thesis is motivated by the need to develop energy-efficient, flexible architectures that can readily adapt to the evolving requirements of modern edge computing workloads. By combining near-memory computing with advanced CPU microarchitecture techniques, this thesis aims at providing the basic blocks for the definition of a new class of edge-oriented embedded systems with improved computational throughput, resource utilization, and energy efficiency, all while leveraging the flexibility and programmability of traditional microprocessors and solely relying on the existing digital design and implementation flow. The proposed solutions, NM-Carus and LEN5, are designed to be scalable and easy to integrate into existing edge computing platforms, providing a seamless upgrade path for future systems that require a balance between performance, energy efficiency, and flexibility.

1.3 Contributions and Publications

This thesis presents several contributions in the field of energy-efficient microarchitectures for low-power edge computing systems. These contributions can be grouped into two primary areas: the design and implementation of a programmable near-memory architecture for integration alongside conventional SRAM banks in small Systems-on-Chip (SoCs), and the design and implementation of a modular and dynamically scheduled RISC-V CPU core optimized for extension with tightly coupled coprocessors.

Although the integration of these two architectures into a single system could not be completed during the author's Ph.D. studies, it remains the ultimate goal of his research. This integration aims to provide a software- and implementation-friendly approach for deploying computationally intensive workloads on edge devices, leveraging a compute-centric memory architecture to accelerate data-driven tasks while exploiting the available instruction-level parallelism in the application control flow through a dynamically scheduled CPU core. This vision motivated the exploration of both architectures and represents the core of the author's current research activities, which are briefly introduced at the end of this manuscript in Section 4.2.

As a side contribution, the author supervised the integration of approximate arithmetic operationsboth software and hardwareinto an existing RISC-V core, within a framework for approximation-aware retraining of ANNs for energy-efficient deployment on CPU-based embedded systems. Each of these areas addresses different challenges in modern edge computing, providing both architectural innovations and practical implementation solutions for immediate exploitation in real-world systems.

The publications resulting from the work conducted during the author's Ph.D. studies are listed below, grouped by topic and type of contribution.

Programmable Near-Memory Computing Architecture

- **Submitted journal article:** M. Caon, C. Choné, P. D. Schiavone, *et al.*, "Scalable and risc-v programmable near-memory computing architectures for edge nodes," 2024. arXiv: 2406.14263 [cs.AR].

Modular, Dynamically Scheduled RISC-V CPU Core

- **Journal article:** M. Caon, G. Masera, and M. Martina, "Ditching the queue: Optimizing coprocessor utilization with out-of-order cpus on compact systems on chip," *Electronics*, vol. 13, no. 15, 2024.
- **Conference paper:** M. Caon, V. Petrolo, M. Mirigaldi, *et al.*, "Seeing beyond the order: A LEN5 to sharpen edge microprocessors with dynamic scheduling," in *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions*, 2024, pp. 47–50.

Approximate ANN Inference on Embedded Microcontrollers

- **Journal article:** F. Guella, E. Valpreda, M. Caon, *et al.*, "Marlin: A co-design methodology for approximate reconfigurable inference of neural networks at the edge," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 71, no. 5, pp. 2105–2118, 2024.
- **Conference paper:** F. Guella, E. Valpreda, M. Caon, *et al.*, "Temet: Truncated reconfigurable multiplier withăerror tuning," in *Applications in Electronics Pervading Industry, Environment and Society*, Cham, 2024, pp. 370–377.

The following sections provide a brief summary of the main contributions published in the aforementioned works, highlighting the key aspects that are discussed in detail in the subsequent chapters of this thesis. Although the LEN5 architecture was developed prior to NM-Carus, this manuscript presents NM-Carus first to enhance the narrative flow, as it effectively motivates many of the design choices made in LEN5's development.

1.3.1 Programmable Near-Memory Computing Architecture

The major contribution of this thesis is the development of a novel NMC approach that bridges the gap between conventional memory hierarchies and the CIM paradigm. The proposed NM-Carus architecture, developed in Lausanne in collaboration with EPFL, targets performance-oriented edge embedded systems, where energy efficiency must be balanced with computational throughput and flexibility. NM-Carus is a fully autonomous, vector-capable, RISC-V-based programmable NMC unit designed for highly parallel and complex TinyML [9] applications such as Deep Neural Networks (DNNs) inference and signal processing. NM-Carus combines a scalable Vector Processing Unit (VPU), an arbitrarily large SRAM-based Vector Register File (VRF), and an embedded-class RISC-V core to maximize flexibility while maintaining a reduced area overhead compared to conventional SRAM memories. NM-Carus's memory-like interface is designed to be compatible with the bus interconnect and memory subsystem of conventional SoCs, where its VRF directly maps part of the host's address space. The result is a *smart* memory that can be programmed with a familiar SIMD ISA, heavily inspired by the RISC-V "V" vector extension [27], and carefully designed to provide a convenient vector-based view of the host's compute memory without incurring the overhead of explicit data movements. In particular, the main contributions of this work are:

- A software-centric, low-implementation-effort, and scalable approach to NMC in the context of general-purpose, low-power edge devices, implemented as a fully autonomous CPU-based programmable memory unit called NM-Carus.
- A novel, vector-based RISC-V ISA extension to support a versatile set of vector operations on programmable CIM architectures. Combined with NM-Carus's memory-like interface, the proposed ISA offers a convenient way to instruct the host system's memory subsystem to execute arbitrary workloads.
- An in-depth, vertical analysis of the impact and benefits of replacing conventional SRAM memory banks with the proposed NMC macros in real-world low-power MCUs.

At about twice the area of a conventional SRAM bank of the same size when implemented on a low-power 65 nm technology node, NM-Carus enables up to 50.0 × lower execution time and 33.1 × higher energy efficiency at the system level compared to executing the same task on a state-of-the-art RISC-V CPU core. NM-Carus reaches its peak energy efficiency of 306.7 GOPS/W when executing 8-bit matrix multiplication operations, surpassing recent state-of-the-art in- and near-memory computing circuits. Its focus on programmability and ease of integration represents a significant advancement over existing CIM architectures, addressing the challenges that prevent them from achieving broader adoption, as clearly outlined in [28]. As a result, NM-Carus offers a seamless upgrade path for existing SoCs and a more flexible and area-efficient alternative to conventional accelerators.

This work is currently under review for publication in a top-tier journal in the field of emerging computing strategies. The preprint version of the manuscript is already available on arXiv in [22].

1.3.2 Modular, Dynamically Scheduled RISC-V CPU Core

A second contribution of this thesis is the design, implementation, and performance characterization of the open-source LEN5 RISC-V microprocessor. LEN5 is a modular, highly configurable, 64-bit RISC-V core featuring a dynamically scheduled execution engine and a speculative front-end. LEN5 was developed to address the challenge of efficient utilization of tightly coupled coprocessors, which often implement long-latency instructions in modern edge-oriented embedded systems. The core is designed to provide high configurability of its control structures, allowing the user to tune the core area and IPC to meet the requirements of the target use case. The execution engine implements latency-independent reservation stations that work as versatile bridges between the speculative, out-of-order core and arbitrary execution units. As a result, LEN5 can be easily extended with custom instructions implemented by tightly coupled coprocessors. The dynamically scheduled pipeline handles a configurable number of in-flight instructions, exploiting the available ILP in the program to maximize the overall utilization of processing resources and therefore the system throughput. LEN5's performance was evaluated using a set of representative benchmarks, including workloads offloading long-latency instructions to custom coprocessors, and compared against state-of-the-art in-order RISC-V cores. The main contributions of this work are:

- The design and implementation of a modular, dynamically scheduled RISC-V core, called LEN5, optimized for the efficient utilization of tightly coupled coprocessors in edge-oriented embedded systems.

- A general strategy for evaluating the benefits of out-of-order instruction execution in systems with tightly coupled coprocessors. The proposed evaluation framework exploits a runtime-configurable dummy coprocessor and an automated test application generator to simulate different instruction mixes and dependency patterns, providing insights into the core's performance and application scalability.

When coupled with long-latency coprocessors, LEN5 achieves up to twice the IPC of state-of-the-art embedded-class RISC-V cores, while maintaining similar performance on standard scalar workloads. Because of its configurable architecture, LEN5's timing performance can be tailored to match the requirements of the target application and technology, showing up to a 30 % uplift in the operating frequency compared to state-of-the-art in-order RISC-V cores when integrated into an open-source, low-power SoC.

The LEN5 core architecture and its performance evaluation when dealing with long-latency coprocessors were published in [24] and [23], respectively. The Register Transfer Level (RTL) description of the core is available at: https://github.com/vlsi-lab/len5.

In performance-oriented SoCs, the out-of-order execution engine of LEN5 effectively complements designs that execute complex computational kernels, such as NM-Carus. By allowing independent instructions to be dispatched and executed while only stalling those with data dependencies on results from offloaded computations, LEN5 maximizes instruction-level parallelism. Although LEN5 can exploit available ILP opportunities when coupled with memory-mapped accelerators, it performs best when paired with tightly coupled coprocessors, where synchronization between the host CPU and the accelerator is handled at the instruction level rather than relying on platform-dependent drivers.

To this end, Section 4.2 presents a vision for integrating multiple instances of NM-Carus within the cache hierarchy of a system using LEN5 as the host CPU. This approach differs from that proposed in [22] by leveraging an instruction-level abstraction of the kernel offloading mechanism instead of relying on driver software and hardware interrupts to manage the execution of offloaded tasks and synchronization between the host and the accelerator.

In such a system, the cache controller implicitly handles hazards between scalar memory instructions and offloaded operations that modify the content of system memory. This allows dependent memory instructions to be stalled while independent ones are executed out of order, unlocking both the full potential of LEN5 in exploiting available ILP and the performance and energy benefits of a NM-Carus-based in-cache compute engine.

1.3.3 Approximate ANN Inference on Embedded Microcontrollers

In addition to the two primary contributions mentioned in the previous sections, the author also supervised the integration of approximate arithmetic operations into an existing RISC-V core in the context of the MARLIN framework [25], with the goal of increasing the energy efficiency of ANN inference on embedded microcontrollers based on the PULP Platform [29]. Since this work is only marginally related to the main contributions of this thesis, it is not discussed in detail in this document. Instead, a brief description of the work is provided below, while the reader is referred to the original publications [25], [26] for further details.

MARLIN is a co-design framework that leverages approximate multipliers to reduce inference energy while minimizing the impact on accuracy by performing layer-wise retraining of the ANN model. The 9-bit multiplier architecture, based on a Dadda reduction tree [30] and detailed in [26], supports 256 approximation levels, including an exact configuration, to offer fine-grained control over the energy and accuracy trade-off. At the highest approximation level, the energy consumption is reduced by 60 %.

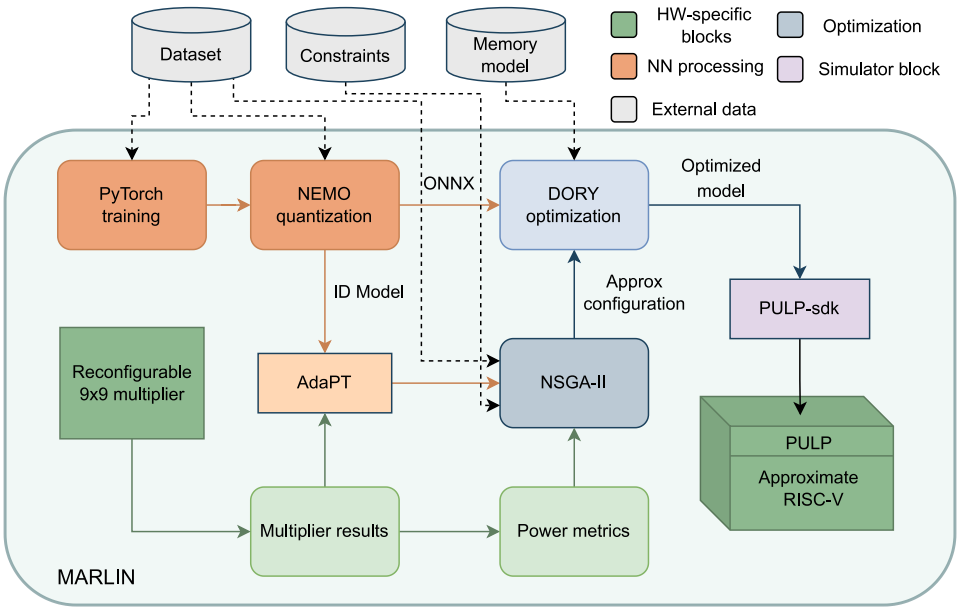


Figure 1.1: MARLIN framework when integrated with the PULP SoC. Source: [25].

In the proposed implementation on the PULP Platform ecosystem, shown in Fig. 1.1, the approximate multipliers were integrated into the Arithmetic Logic Unit (ALU) of the microcontroller's RISC-V core [31], with a 7.8 % area overhead and no impact on the core's timing characteristics. The approximation level is controlled using a dedicated Control and Status Register (CSR), whose access instructions are automatically inserted in the generated inference code by the framework through a modified version of the PULP deployment toolchain,

allowing for seamless deployment of the approximate model on the target platform. MARLIN leverages the NSGA-II genetic algorithm [32] to search for optimal configurations among the possible layer-wise approximate network models. The network accuracy and the estimated power consumption, based on post-synthesis simulations, are used as optimization objectives.

When tested on the CIFAR-10 dataset with different configurations of the ResNet model [33], MARLIN can achieve up to 20 % energy savings while keeping the network accuracy within 1 % of the exact model.

MARLIN is open-source and available at: https://github.com/vlsi-lab/MARLIN.

1.4 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 presents NM-Carus, a novel near-memory processing architecture that enables efficient and flexible in-memory computation. Chapter 3 introduces LEN5, an out-of-order RISC-V core designed for energy-efficient edge computing. Finally, Chapter 4 concludes the thesis, summarizing the contributions and discussing ongoing and potential future research in the field of programmable near-memory computing.

Chapter 2

NM-Carus: Programmable Near-Memory Computing

2.1 Limitations of Current Compute-In-Memory Architectures

The rise of data-centric workloads in edge computing has exposed the inefficiencies of traditional von Neumann architectures, especially when handling memory-intensive operations. The continuous data transfer between memory and processing units, also known as the von Neumann bottleneck, results in high energy consumption and performance limitations. To address these issues, a range of alternative architectures, including domain-specific accelerators and CIM systems, have been proposed [17].

Among these solutions, CIM architectures have garnered significant attention for their ability to reduce data movement by processing data directly within or near the memory arrays. There are two primary aspects contributing to the reduction in energy consumption:

- An architectural advantage dictated by the possibility to directly access source and destination operands from the memory array, without the need for explicit memory transfer instructions to move data between the memory and the processor's General-Purpose Registers (GPRs). This feature reduces both the number of instruction fetches (i.e., no *load* or *store* instructions) and the amount of data transfers.
- A technological advantage that stems from the reduced energy consumption of accessing the data, which does not traverse the system bus and, possibly, lower levels of the

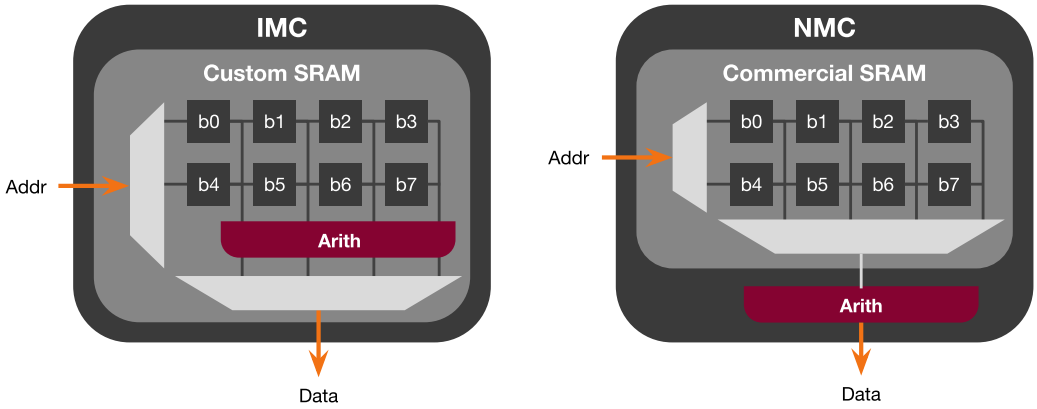


Figure 2.1: Architectural differences between IMC and NMC solutions. In IMC devices, the processing units are embedded within the memory arrays, so accessing the stored data does not require traversing the array periphery. In NMC systems, the processing units are placed near the memory arrays, yet outside the periphery circuitry.

memory hierarchy. This advantage is particularly relevant in modern memory technologies or large SoCs, where the energy consumption of data transfers is a significant portion of the overall system energy budget.

However, despite their promising advantages in terms of energy efficiency and performance, existing CIM architectures face several limitations that hinder their broader adoption, especially in low-power, programmable edge devices.

For the purpose of the discussion in this chapter, two types of CIM architectures are identified: In-Memory Computing (IMC) and Near-Memory Computing (NMC) systems. As depicted in Fig. 2.1, IMC architectures embed processing units within the memory arrays, while NMC architectures place the processing units near the memory arrays, yet outside the periphery circuitry. As a consequence, IMC solutions offer superior energy efficiency when accessing the stored data, as the memory periphery is bypassed. However, they suffer from limitations in terms of reliability, precision, and technological portability, with direct implications on development and implementation costs and time to market. In contrast, NMC solutions provide a more practical and scalable approach, as they separate the memory and compute elements, allowing for more straightforward integration into existing SoCs and digital design flows [17], [34].

2.1.1 In-Memory Computing Limitations

IMC architectures integrate processing units directly into the memory arrays, exploiting the internal structure and parallelism of the memory cells to achieve superior arithmetic throughput with lower area overhead compared to memory-mapped accelerators [35]. In particular, emerging Non-Volatile Memory (NVM) technologies like Magnetoresistive Random Access Memory (MRAM), Phase-Change Memory (PCM), and Resistive Random Access Memory (RRAM) have demonstrated potential for energy-efficient matrix multiplications in analog computing domains [36]–[38]. These technologies are well-suited for applications like ANN, where some inaccuracy is tolerable [39]. However, their stochastic behavior presents significant challenges in terms of reliability, precision, and portability, rendering them unsuitable for applications requiring strict accuracy and memory integrity [39].

Moreover, SRAM-based IMC architectures, while providing fine-grained optimization within existing semiconductor processes, suffer from severe data-placement constraints and reduced scalability. Embedding processing elements directly into the memory array complicates the memory hierarchy, limiting data flexibility and requiring customized application software. These issues make SRAM-based IMC solutions invasive and disruptive at the system level, leading to performance degradation in complex, data-dependent workloads and hindering their integration into standard SoC architectures.

2.1.2 Near-Memory Computing as a Practical Solution

Near-Memory Computing architectures, which place digital processing units near but outside the memory arrays, offer a more practical and scalable solution. By separating the memory and compute elements, NMC systems avoid the limitations of IMC in terms of data placement and system integration. This separation also allows NMC designs to rely on standard digital design flows, making them more portable and easier to implement across different technology nodes [17], [34].

The proposed NM-Carus architecture adopts the NMC approach by combining an energy-efficient SRAM-based memory system with a programmable, RISC-V-based processing unit. Unlike conventional accelerators that require specialized local memory and complex software management [40], [41], NM-Carus can be seamlessly integrated into existing SoCs as a drop-in replacement for conventional SRAM banks. This reduces the need for data movement and enables direct processing within the memory subsystem, significantly improving energy

efficiency and performance in data-intensive applications like DNN inference and signal processing [42].

2.1.3 The Need for Programmable, SRAM-Based CIM Solutions

Programmability is a key factor in the design of NM-Carus, as it allows the memory subsystem to adapt to a wide range of applications without sacrificing flexibility. Many fixed-function accelerators and IMC devices are limited in scope, designed to execute specific operations efficiently but unable to handle diverse workloads [43], [44]. In contrast, NM-Carus supports a versatile ISA based on the RISC-V vector extension, enabling the execution of a variety of SIMD operations directly in memory. This programmability offers the best of both worlds: the energy efficiency of CIM and the flexibility of traditional microprocessors.

Furthermore, NM-Carus improves upon earlier designs like NM-Caesar [22], which relied on command streams from the host system to execute CIM operations [34]. While NM-Caesar achieved impressive energy efficiency by using a micro-controlled architecture, it lacked support for vector operations and had lower throughput for complex tasks. In contrast, NM-Carus features a fully autonomous processing unit, capable of executing more complex and data-parallel operations without constant intervention from the host CPU. This autonomy reduces the overhead of control and improves system performance by allowing the CPU to handle other tasks in parallel.

The innovative integration model presented by NM-Carus also addresses many of the challenges faced by current CIM solutions. By maintaining a conventional SRAM interface, NM-Carus minimizes the software and hardware integration effort, allowing it to be easily adopted in existing SoCs without requiring major changes to the software stack or physical design. This approach strikes a balance between energy efficiency, programmability, and ease of implementation, making NM-Carus a practical solution for modern edge computing applications that demand both high performance and flexibility.

In conclusion, while traditional CIM architectures offer substantial improvements in energy efficiency for specific applications, their limitations in terms of scalability, flexibility, and system integration prevent them from being widely adopted in general-purpose edge computing systems. NM-Carus overcomes these limitations by providing a programmable, scalable, and energy-efficient solution based on standard SRAM technology, making it a practical and deployable option for a wide range of data-intensive applications in energy-constrained environments.

2.2 Proposed Near-Memory Computing Architecture

This section explores the architecture of the proposed NM-Carus, detailing its integration into existing systems and the design considerations influenced by application requirements and physical constraints.

2.2.1 Design Philosophy

The accelerating computational demands of modern edge computing devices necessitate innovative solutions that can be seamlessly integrated into current SoCs with minimal effort. The primary goal of the proposed NMC paradigm is to deliver the efficiency of CIM to these systems, addressing the burgeoning computational needs without extensive redesigns. To achieve this, the NMC device must satisfy two key requirements:

1. **Functional Integration:** The device should integrate into the host system's memory space, behaving like conventional memory during regular operations;
2. **Physical Compatibility:** It must serve as a direct substitute for a standard SRAM memory bank, matching its area footprint and timing characteristics.

An illustration of the integration model fulfilling these requirements is provided in Fig. 2.2.

System Integration Model

To ensure seamless functional integration 1, the NMC device connects to the system bus using a slave interface, similar to standard memory components. It operates in two primary modes:

- *Memory Mode*: Offers standard read and write access to memory contents, allowing the host CPU or Direct Memory Access (DMA) engine to manage data storage and retrieval.
- *Computing Mode*: Enables the execution of processing tasks directly within the device, either by streaming instructions or programming processing kernels.

By performing computations within its own memory, the NMC device reduces data movement overhead, and results are readily accessible by the host system. Communication occurs

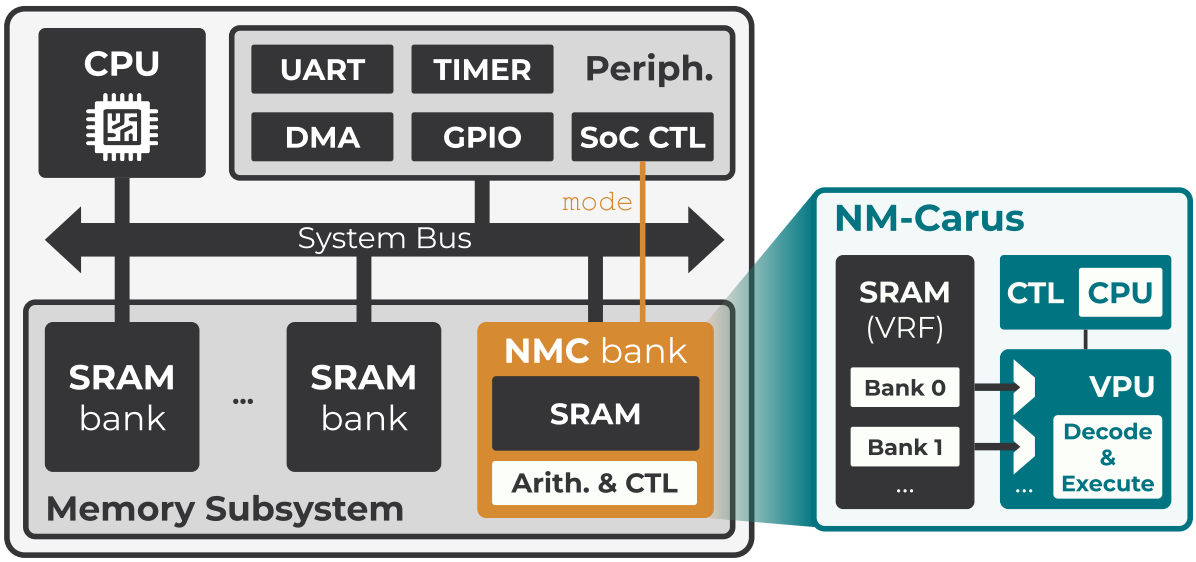


Figure 2.2: Top-level block diagram of an NMC-enhanced MCU incorporating the proposed NM-Carus as part of its memory subsystem. This setup contrasts with traditional methods that rely on streaming commands over the bus interface. Source: [22].

via a memory-like interface; when in computing mode, the data write bus is utilized to send instructions. A dedicated `imc` pin, controlled by a configuration register within the host system, switches between modes without modifying the bus protocol. This approach avoids address space fragmentation, a common issue in methods where the address bus encodes instructions, thereby simplifying firmware development and memory mapping.

The overhead associated with toggling the `imc` configuration is minimal and acceptable in typical edge-computing scenarios where long processing sequences occur between memory access operations.

Physical Implementation Considerations

Addressing the physical compatibility requirement 2, the NMC device is designed using standard digital implementation flows and utilizes foundry-provided SRAM compilers for its internal memory arrays. This methodology enhances portability across different technologies compared to IMC solutions that depend on custom memory designs. The control logic and data processing units are engineered to match the timing profiles of conventional SRAM memories of equivalent capacity, minimizing area overhead. Maintaining consistent input and output delays ensures that the NMC device can replace standard memory banks without necessitating alterations to the surrounding system architecture.

Design Motivation and Architecture Overview

With the integration model defined, it is essential to discuss the design motivations that shape the NM-Carus architecture. The fully digital NMC paradigm allows for the creation of scalable and configurable architectures that can meet varying performance demands. However, ultra-constrained IoT devices often face physical limitations that restrict the level of flexibility achievable, which is more beneficial in higher-performance systems.

Traditional approaches commonly involve the host system streaming SIMD instructions over the bus interface, encoding commands within the data fields of bus transactions. While this method minimizes additional hardware and power consumption, it can suffer from limited flexibility and reduced performance due to the overhead of continuous command streaming and potential bus contention.

The proposed NM-Carus architecture overcomes these limitations by integrating a RISC-V-based controller that supports a custom CIM-oriented ISA. This controller is coupled with a novel and scalable VPU, providing enhanced flexibility and adjustable data-level parallelism. By offloading complex processing tasks to the NMC device itself, NM-Carus reduces dependency on the host CPU and minimizes bus traffic, leading to improved efficiency and performance.

In contrast to methods that rely heavily on the host system for instruction streaming, NM-Carus offers a more autonomous solution. Its integrated controller allows for sophisticated processing capabilities within the NMC device, enabling it to handle a broader range of workloads without constant intervention from the host.

The subsequent sections will provide an in-depth examination of the NM-Carus architecture, including its instruction set and microarchitectural details. This exploration will highlight how the design meets the goals of enhanced performance, flexibility, and ease of integration, making it a compelling solution for next-generation edge computing devices.

Building upon the integration model established earlier, the architecture of NM-Carus is now detailed. NM-Carus is tailored for edge devices that can afford a slightly larger area footprint in exchange for enhanced performance and versatility alongside energy efficiency.

Design Objectives

The NM-Carus architecture is engineered to harness the increased data-access efficiency provided by the NMC paradigm while achieving the following key objectives:

1. **Enhanced Flexibility and Scalability:** Offering the advantages of programmable vector machines, including flexibility, scalability, and reduced instruction fetch overhead;
2. **Configurable Parallelism:** Employing a scalable microarchitecture that allows for configurable data-level parallelism.

Architectural Overview

To fulfill objectives 1 and 2, NM-Carus integrates an innovative controller based on a compact RISC-V CPU, referred to as the embedded CPU (eCPU) to distinguish it from the host system's processor. Positioned close to the internal SRAM banks, the eCPU manages computation within the NMC device. A dedicated memory, the embedded Memory (eMEM), is programmed by the host system with the CIM kernels to be executed and provides minimal support for stack and heap management.

The eCPU offloads vector instructions utilizing a custom `xvnmc` ISA extension (detailed in Section 2.2.2) to a novel, area-efficient, and scalable VPU (described in Section 2.2.3). This arrangement contrasts with traditional approaches that rely on streaming commands over the busembedding instructions within data fieldswhich can lead to bus contention and increased overhead. By integrating the controller and vector processing capabilities within the memory component, NM-Carus reduces dependence on the host CPU and minimizes bus traffic.

Notably, the NM-Carus controller is fully programmable and Turing-complete, compatible with any RISC-V-compliant software build toolchain. Support for the `xvnmc` instruction set is currently provided at the assembler level within the GNU Compiler Collection (GCC). To the best of our knowledge, NM-Carus represents the first instance of a software-programmable, autonomous embedded memory device that supports vector operations through a newly introduced NMC-specific RISC-V ISA extension.

System Perspective and Integration

Despite incorporating a CPU within its architecture, NM-Carus is perceived as a memory component from the system's standpoint, adhering to the same integration model discussed earlier. As elaborated at the beginning of this Chapter, the absence of integrated load-store units and master ports in its external interface differentiates NM-Carus from conventional memory-mapped accelerators.

Moreover, a key distinction between NM-Carus and isolated vector coprocessors with private memories lies in the emphasis on memory capacity over processing resources. The VRF,

which accounts for at least half of NM-Carus's die area, is memory-mapped from the system's perspective and serves as the sole data source for the internal VPU. In contrast, traditional vector units interact with external memories for data loading and storage, offering greater flexibility in VRF data layout but incurring increased data movement overhead and associated energy costs.

By centralizing data storage within the VRF and minimizing external data transfers, NM-Carus optimizes performance and energy efficiency for data-intensive workloads typical of edge computing applications.

Advantages Over Near-Memory CPU Cores

Traditional NMC implementations often rely on the host system to stream commands over the bus interface, encoding instructions within data transactions. While this method reduces additional hardware requirements, it suffers from limitations in flexibility and performance due to bus contention and the overhead of continuous command streaming.

One might consider introducing one or more regular CPU cores near the memory banks to achieve flexibility and programmability similar to NM-Carus, while saving the area overhead of the VPU. However, this approach does not effectively address the inefficiencies associated with data movement and instruction fetching inherent in conventional CPU architectures. A traditional CPU must explicitly load data from memory into its internal registers using load instructions, perform computations, and then store the results back to memory using store instructions. This process results in significant data movement overhead and increased energy consumption due to the numerous load and store operations and the need to fetch and execute many instructions for scalar computations.

In contrast, NM-Carus offers a more efficient solution by using a minimal RISC-V RV32E CPU[1] as a Turing-complete controller that simply implements the kernel control flow and offloads vector computing instructions to an area-efficient VPU. The VPU operates directly on the data stored in memory, which is treated as a large vector register file (VRF), allowing it to perform computations without the need for explicit data movement between memory and processor registers. This eliminates the overhead of load and store instructions and reduces data movement, leading to improved energy efficiency. Furthermore, the VPU in NM-Carus is inherently SIMD. This means that a single vector instruction performs operations on wide

[1] Since all computation is offloaded to the VPU, the controller CPU does not integrate a hardware multiplier or divider and has an Register File (RF) with 16 registers instead of 32 as in the base RV32I ISA. This helps contain the area overhead of integrating a CPU-based controller.

vectors comprising hundreds or thousands of elements[2], without the need for explicit branch instructions to loop over the vector elements. As a result, the number of instructions that need to be fetched and executed is significantly reduced compared to a scalar CPU. This reduction in instruction fetch overhead contributes to lower energy consumption and smaller code size, enhancing the overall efficiency of the system.

Therefore, compared to simply introducing a regular CPU core near the memory bank, NM-Carus offers substantial advantages in terms of reduced data movement, decreased instruction fetch overhead, and enhanced energy efficiency, all while providing a programmable and flexible computing solution integrated within the memory hierarchy. By integrating the controller and vector processing capabilities within the memory component, NM-Carus reduces dependence on the host CPU and minimizes bus traffic. The result is an architecture that balances increased computational capabilities with energy efficiency and ease of integration into existing systems.

2.2.2 Instruction Set Architecture

Overview of the `xvnmc` Extension

The `xvnmc` is a custom vector extension for the RISC-V architecture, specifically developed to enable small and efficient NMC devices to handle parallel workloads effectively while minimizing hardware implementation costs (see Section 2.3). Drawing significant inspiration from the standard RISC-V Vector Extension (RVV) [27], the `xvnmc` shares many instruction formats and semantics with RVV. The built-in support of RVV for stripmining and large vectors (up to 64 KiB) aligns well with the scalable, memory-centric microarchitecture of NM-Carus, allowing code compatibility across implementations with varying VRF sizes.

Instruction Set Details

The custom instructions introduced in `xvnmc`, summarized in Table 2.1, include integer single-width arithmetic, logic, and shift operations, as well as vector permutation instructions analogous to their RVV counterparts. Additionally, `xvnmc` introduces two unique instructions, `xvnmc.emvv` and `xvnmc.emvx`, specifically designed to facilitate data exchange between an arbitrary element of a vector register and a scalar general-purpose register (GPR) in the eCPU. This mechanism is the sole method by which the eCPU code can interact with the contents

[2]The vector length depends on the configured size and number of memory banks inside the NM-Carus instance.

of the VRF, which is not accessible through standard load and store instructions. This design choice implicitly isolates access to the memory banks from the scalar code running on the eCPU, ensuring that synchronization and sub-word data access are handled exclusively by the VPU hardware according to its current status and configuration. Additionally, it ensures that no slow combinational paths are introduced through the memory bank periphery and the eCPU's load and store unit, which could limit the maximum operating frequency of the system. Consequently, the eCPU's instructions and private data are stored in the embedded memory (eMEM) instead. Since the NM-Carus VPU operates directly within the host system's memory space and relies on the system to populate its VRF, vector load and store instructions are unnecessary.

Table 2.1: `xvnmc` instruction listing. Instruction variants are expanded in Table 2.2. Vector arithmetic operations are element-wise unless otherwise specified. Instruction with the `[r]` type can optionally use indirect register addressing.

Mnemonic	Variants			Description
Vector Integer Arithmetic-Logic Instructions				
`xvnmc.vadd[r]`	vv	vx	vi	Addition
`xvnmc.vsub[r]`	vv	vx		Subtraction
`xvnmc.vmul[r]`	vv	vx		Multiplication
`xvnmc.vmacc[r]`	vv	vx		Multiply-Add
`xvnmc.vand[r]`	vv	vx	vi	Bitwise AND
`xvnmc.vor[r]`	vv	vx	vi	Bitwise OR
`xvnmc.vxor[r]`	vv	vx	vi	Bitwise XOR
`xvnmc.vmin[u][r]`	vv	vx		Select minimum
`xvnmc.vmax[u][r]`	vv	vx		Select maximum
`xvnmc.vsll[r]`	vv	vx	vi	Logic shift left
`xvnmc.vsrl[r]`	vv	vx	vi	Logic shift right
`xvnmc.vsra[r]`	vv	vx	vi	Arithmetic shift right
Vector Permutation Instructions				
`xvnmc.vmv[r]`	vv	vx	vi	Copy/splat into vector
`xvnmc.vslideup[r]`	vx	vi		Slide elements up
`xvnmc.vslidedown[r]`	vx	vi		Slide elements down
`xvnmc.vslide1up[r]`	vx			Slide up and push GPR
`xvnmc.vslide1down[r]`	vx			Slide down and push GPR
Scalar-Vector Movement Instructions				
`xvnmc.emvv`	ex			Move GPR to v[i]
`xvnmc.emvx`	xe			Move v[i] to GPR
Configuration Instructions				
`xvnmc.vset[i]vl[i]`	N/A[a]			Set VL and SEW

[a] Reserved formats, same as RVV.

Rationale Behind Instruction Set Design

Synchronization within the vector pipeline is ensured by its single-issue microarchitecture, where the coprocessor generates target element addresses based on the element index, target

Table 2.2: NM-Carus instruction formats. The `xvnmc` extension is implemented inside the RISC-V *Custom-2* 25-bit encoding space with the `0x5b` major opcode. The standard OPIVV, OPIVX, and OPIVI formats are used for the `vv`, `vx`, and `vi` instruction variants respectively, while `ex` and `xe` use the OPMVX format. Indirect register addressing is available for all `vv`, `vx`, and `vi` instructions.

Var.	Dest.	Data source(s)		Example
vv	`vd[:]`	`vs2[:]`	`vs1`	`xvnmc.vadd.vv v2, v1, v0`
vx	`vd[:]`	`vs2[:]`	`rs1`	`xvnmc.vadd.vx v2, v1, x5`
vi	`vd[:]`	`vs2[:]`	`imm`	`xvnmc.vadd.vi v2, v1, 42`
ex	`vd[rs2]`	`-`	`rs1`	`xvnmc.emvv v0, x4, x5`
xe	`rd`	`vs2[rs1]`	`-`	`xvnmc.emvx x5, v0, x4`

vector specified in the instruction, and the current configuration of vector length and standard element width (Selected Element Width (SEW)). Consequently, vector load and store instructions would primarily be used to exchange data with the small code memory (eMEM) that holds the eCPU's instructions and data. However, utilizing this memory for kernel input or output data would undermine the core objective of the NMC paradigm, which is to compute without moving data between memory devices. This consideration explains the absence of vector load and store instructions in the `xvnmc` instruction set.

Nevertheless, certain workloads cannot be efficiently handled using vector instructions alone, necessitating the eCPU to access the VRF to participate in the computation when required. While this could be facilitated using scalar load and store instructions by connecting the VRF to the CPU's data memory interface, such an approach would:

1. Complicate memory access arbitration,
2. Demand additional hardware to prevent race conditions between vector instructions and scalar load/store operations, and
3. Require scalar software to be cognizant of the VRF's memory layout and vector element width to issue memory accesses with the correct address and data width, leading to increased code size and complexity.

These challenges justify:

1. The omission of vector load and store instructions in the `xvnmc` instruction set, and
2. The physical isolation of the NM-Carus VRF from the internal CPU memory interfaces.

Indirect Register Addressing for Flexibility

A significant challenge when using a vector ISA without vector load and store instructions in the proposed CIM model is that the data layout within the VRF is not determined by the vector kernel. Consequently, to iterate over N chunks (or rows) of input data mapped to N vector registers, N vector instructions with different source and destination registers would be required. With standard vector instructions, this necessitates fully unrolling the iteration process. The situation worsens with nested loops in the kernel, leading to exponential growth in code size. This problem does not arise in traditional vector machines, where data can be loaded from any memory location into the same vector registers during each loop iteration.

In NM-Carus, however, the VRF's data layout is dictated by the host system and may be unknown at compile time, especially if memory allocation is dynamic. As a result, using the same kernel for different subsets of registers becomes infeasible if the source and destination registers are hardcoded as immediate values in the kernel's vector instructions, as is the case with standard RVV.

To overcome these challenges, the `xvnmc` extension introduces instruction variants that support *indirect register addressing*. In these variants, the indices of the source and destination vector registers are encoded in the three least-significant bytes of a scalar general-purpose register (GPR) (`rs2`). This approach supports up to 256 logical vectors, accommodating applications with various data shapes, similar to the method proposed in [45]. Consequently, the same vector instruction can be reused across loop iterations with different operands by simply updating the GPR containing the indices. This dramatically reduces code size and allows the same kernel to be used for input data with different base addresses and iteration counts.

This mechanism is analogous to updating the base address in load or store instructions within a conventional software loop and can be easily integrated into a custom compiler. Moreover, since a single GPR encodes all instruction operands, updating indices requires only a single `add` instruction, further reducing code size. The execution time for index-updating instructions is typically masked by the latency of the last issued vector instruction, thanks to NM-Carus's ability to execute scalar and vector instructions in parallel. The energy overhead of index management instructions is negligible compared to vector operations, as evidenced by the minimal energy contribution of the eCPU shown in Fig. 2.11.

Additionally, the omission of vector load and store instructions renders the `xvnmc` ISA completely independent of data bitwidth, enhancing reusability compared to RVV.

Bridging Memory and Register Views

In summary, the proposed `xvnmc` ISA provides a software-friendly, architecture-agnostic interface that leverages the CIM model, effectively bridging the host system's *memory* perspective of the CIM device with an internal *register* perspective. By employing vector processing, it enhances data-level parallelism and energy efficiency. The use of indirect vector register addressing contributes to reduced code size and increased reusability compared to conventional vector ISAs.

The inclusion of indirect register addressing and arbitrary vector-to-scalar element movements is particularly motivated by the dual role of NM-Carus. Serving both as a part of the system memory space for the host software and as a vector processor for its own kernel code, NM-Carus leverages the enhanced energy efficiency enabled by computing-in-memory. Designed to be a drop-in, *smarter* replacement for the lowest level of the *memory* hierarchy in small, low-power *edge devices*, it aims to match the area, timing, and memory capacity characteristics of a traditional SRAM bank (see Section 2.3).

This design philosophy results in a VRF with a larger capacity than that of typical vector machines, allowing for larger data tiles to be acquired and processed directly without the need for data movementan inherent advantage of IMC and NMC implementations. The `xvnmc` instructions thus provide a software-friendly method to bridge the *memory* and *register* views of the same data from both system and computing kernel perspectives, while retaining the benefits of vector processing in terms of data-level parallelism and energy efficiency.

Practical Example: Matrix Multiplication

To illustrate the capabilities of the `xvnmc` extension, consider the implementation of a matrix multiplication kernel ($R = A \times B$). In NM-Carus, the matrices are stored within the VRF, and computations are performed using the custom vector instructions. An example kernel is shown in Listing 1, where custom `xvnmc` instructions are highlighted in bold.

In contrast, a traditional RVV-based implementation would require explicit load and store instructions, increasing code size and data movement overhead. An example of such a kernel is presented in Listing 2, where RVV instructions are highlighted in bold.

These examples highlight how the `xvnmc` extension allows for efficient implementation of parallel algorithms while minimizing code size and data movement, thereby enhancing performance and energy efficiency in NMC devices like NM-Carus.

NM-Carus Matrix Multiplication (Any Size, Any Bitwidth)

```
_start:
  # ... initialization code
  # - A is flattened in v31
  # - B is available in v0-v14
  # - R is stored in v15-v30
L0:
  xvnmc.emvx s1, v31, t0     # Load A[t0] (v31[t0])
  addi       t0, t0, 1       # Increment A element pointer
  xvnmc.vmaccr.vx t1, s1     # R[m,:] += A[m,n] * B[n,:]
  addi       a3, a3, 1       # Increment B row counter
  addi       t1, t1, 0x0100  # Increment B row pointer (vs2++)
  bltu       a3, a1, L0      # Iterate over the rows of B
  andi       t1, t1, 0x00ff  # Reset B row pointer (vs2=v0)
  addi       t1, t1, 0x0001  # Update B and R (vs2=v1; vd++)
  addi       a2, a2, 1       # Increment R row counter
  li         a3, 0           # Reset B row counter
  bltu       a2, a0, L0      # Iterate over the rows of R
  ret                        # Return
```

Listing 1: Example of a NM-Carus kernel for matrix multiplication ($R = A \times B$) using `xvnmc` instructions. Custom `xvnmc` instructions are highlighted in bold.

The comparison with the same algorithm implemented in RISC-V assembly for a scalar CPU is omitted here, while the code is provided in Section A.1 for reference.

2.2.3 Microarchitecture

2.2.4 Microarchitecture

An overview of the NM-Carus microarchitecture is illustrated in Fig. 2.3.

Controller Architecture

The controller of NM-Carus operates as a minimal System-on-Chip (SoC), integrating the OpenHW Group's four-stage, in-order CV32E40X [46] RISC-V corereferred to as the eCPU[3]a compact code memory (eMEM), and a configuration register that manages synchronization

[3]Although smaller RISC-V CPUs are available and might be more suitable for the NM-Carus controller, the CV32E40X was selected due to its CORE-V X interface, which simplifies the implementation of custom coprocessors.

RVV Matrix Multiplication (Any Size, 32-bit)

```
_start:
  # ... initialization code
L0:
  lw       s1, 0(t0)    # Load A[t0]
  addi     t0, t0, 4    # Increment A element pointer
  vle32.v  v0, 0(t1)    # Load B[n,:]
  addi     t1, t1, 4VL  # Increment B row pointer
  vmacc.vx v1, v0, s1   # R[m,:] += A[m,n] * B[n,:]
  addi     a3, a3, 1    # Increment B row counter
  bltu     a3, a1, L0   # Iterate over the rows of B
  vse32.v  v1, 0(t2)    # Store R[m,:]
  addi     t2, t2, 4VL  # Increment R row pointer
  li       t1, B_ADDR   # Reset B row pointer
  li       a3, 0        # Reset B row counter
  addi     a2, a2, 1    # Increment R row counter
  bltu     a2, a0, L0   # Iterate over the rows of R
  ret                   # Return
```

Listing 2: Example of an RVV kernel for matrix multiplication ($R = A \times B$). RVV instructions are highlighted in bold.

with the host system. To reduce the core area, the eCPU implements the RV32EC ISA, providing 16 general-purpose registers (GPRs) and excluding hardware multiplication and division units.

Instructions from the custom `xvnmc` extension are offloaded by the eCPU to the vector processing unit (VPU) via the CORE-V X interface [47]. All components of the controller are interconnected through a single-channel bus, which is exposed to the host as a memory-mapped slave peripheral using the same multiplexed memory-like interface employed to access the NM-Carus data memory.

In *configuration mode*, the host can program the eMEM with kernel code or control kernel execution by accessing the configuration register. During kernel execution, NM-Carus can be switched back to the normal *memory mode*, allowing standard memory operations to proceed (e.g., implementing double buffering). Upon completion of the kernel execution, a dedicated status bit signals the end of computation to the host system. This status bit can also be routed to an optional pin in the NM-Carus top-level interface, serving as an interrupt source to enable the host system to enter a low-power state during computation.

As discussed in Section 2.2.2, the `xvnmc.emvx` instructions are the only ones that introduce data hazards between vector and scalar instructions. Therefore, in most scenarios, scalar and

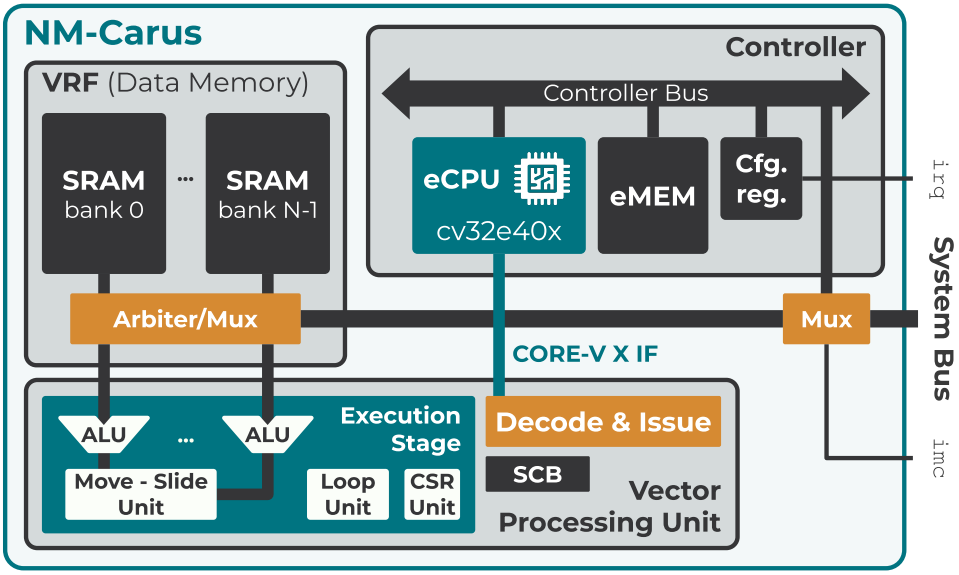


Figure 2.3: Top-level block diagram of NM-Carus. Source: [22].

vector instructions can execute in parallel, as illustrated in Fig. 2.4, thereby reducing kernel execution time.

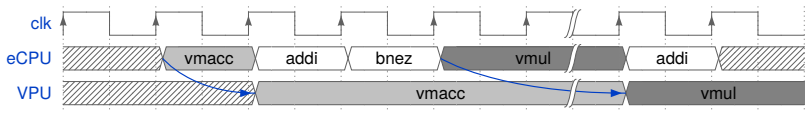


Figure 2.4: Parallel execution of scalar and vector instructions in NM-Carus. Source [22].

Vector Processing Unit Design

The VPU in NM-Carus is designed as a single-issue vector machine with a configurable level of hardware unrolling to accommodate different performance requirements. The pipeline consists of three primary components:

1. A *Decode Stage* that interfaces with the eCPU and dispatches decoded controls and scalar data to the execution units.
2. An *Execution Stage* containing three dedicated units responsible for arithmetic operations, permutation instructions, and control and status register (CSR) instructions.
3. A shared *Commit Unit* that tracks the execution status of in-flight instructions using a minimal scoreboard and handles synchronization with the eCPU.

To conserve energy, the entire VPU is clock-gated when no vector instructions are in execution. The decision to implement a single-issue design stems from the desire to minimize the complexity and size of control logic, arbitration mechanisms, and internal buffers. This approach allows the majority of the area and power budget to be allocated to the arithmetic circuitry, which is critical for the VPU's overall throughput.

While wider or deeper execution pipelinesas seen in high-performance vector machines [48]can enhance the utilization of functional units and data interfaces, they introduce additional hardware complexity and power consumption. In the context of NM-Carus, which is intended to work with a large vector register file (VRF), the idle time of execution units is minimal relative to the total instruction execution time. Therefore, the added cost of more complex pipelines does not yield proportional performance benefits.

Vector Register File Organization

The data memory in NM-Carus is composed of a configurable number of single-port, 32-bit SRAM banks with flexible capacity. The current VRF architecture supports 32 logical vector registers, consistent with the standard RVV. The data interleaving policy adoptedillustrated in Fig. 2.5maps words that are contiguous in the host system's address space to adjacent VRF banks. This alignment ensures that elements with the same index across different vector registers are physically located in the same VRF bank.

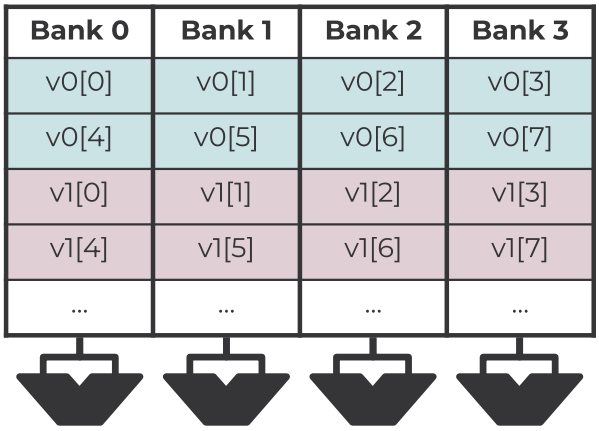


Figure 2.5: Organization of the NM-Carus Vector Register File (VRF). Source: [22].

This configuration facilitates straightforward vector unrolling, as all operands for an element-wise instruction reside in the same bank. Each arithmetic logic unit (ALU) in the arithmetic unit is connected to a single SRAM bank, forming an independent computational *lane*. This design allows the VPU to be scaled arbitrarily: increasing the number of lanes enhances the

level of unrolling, thereby improving throughput at the expense of additional area and power consumption.

Since all operands are located in the same bank, they must be accessed sequentially. However, this does not create a bottleneck because NM-Carus employs compact, serial arithmetic circuits, which offer significant implementation advantages over parallel designs. Achieving the necessary throughput with parallel arithmetic units would require larger, less efficient multi-port memory banks or more complex VRF layouts, introducing additional routing and hazard-detection hardware. Furthermore, the vector instructions supported range from those requiring a minimum of one vector source operand per result (e.g., `xvnmc.vadd.vx`) to those requiring up to three (e.g., `xvnmc.vmacc.vv`). Optimizing memory access patterns to accommodate such a diverse set of operations would complicate the design and exacerbate these issues.

Execution Engine and Functional Units

At the core of the VPU is an execution engine comprising three dedicated execution units:

1. An *Arithmetic Unit* responsible for integer arithmetic and logic instructions.
2. A *Move-Slide Unit* that performs permutation operations and scalar-vector data movements.
3. A *CSR Unit* that manages access to the `vtype` control and status register (CSR), which encodes the current vector length and element bitwidth.

A shared loop unit generates the VRF addresses required to iterate over the elements of the source and destination vectors. The datapaths of units 1 and 2 are interdependent because the ALUs within the arithmetic unit and their internal registers are used to buffer intermediate data from both arithmetic and permutation operations. Resource sharing is extensively utilized within the ALUs to implement packed-SIMD operations with minimal area overhead.

To achieve this, each ALU is equipped with:

- A partitioned, multi-precision 16-bit adder, capable of handling various operand sizes.
- A 16-bit multiplier for arithmetic computations.
- Basic logic operators for bitwise operations.

- A serial 8-bit logic and arithmetic barrel shifter.

For 32-bit addition, two successive 16-bit additions are performed; 32-bit multiplication is computed using three successive 16-bit multiplications accumulated by the adder; 8-bit multiplications are sign-extended to 16 bits and truncated as necessary. Consequently, the adder can process 32-bit data words every two cycles, regardless of the element bitwidth. The multiplier produces:

- Four 8-bit results in four cycles.
- Two 16-bit results in two cycles.
- One 32-bit result in three cycles.

Control Logic and Scheduling

Each ALU executes the same operation on different elements of a vector, effectively implementing data-level parallelism. As a result, all lanes share a common ALU control unit and memory access scheduler. The compute and memory access patterns are designed to ensure that the throughput of the arithmetic unit is never less than that of the slower component between the ALU and the VRF, for any combination of ALU operation and operand bitwidth.

Arithmetic and memory operations are dispatched to the ALU control unit or the VRF by a shared finite state machine (Finite State Machine (FSM)) that employs a combination of static and dynamic scheduling techniques. This allows the system to handle all possible combinations of ALU operation latencies and varying numbers and types of vector and scalar operands. The control unit orchestrates and interleaves memory accesses from successive iterations to ensure that the operational throughput achieved is not less than the minimum attainable by either the ALUs or the VRF for the current instruction.

For example, each lane can produce a 32-bit word from a 16-bit `xvnmc.vmacc.vv` instruction in a maximum of four cycles. Although the ALUs can complete the two 16-bit multiply-accumulate (Multiply-and-Accumulate (MAC)) operations in just two cycles, the throughput is constrained by the four cycles required to fetch the three source operands (`vd[i]`, `vs2[i]`, and `vs1[i]`) and store the result (`vd[i]`).

This careful coordination between computation and memory access ensures that the VPU operates efficiently, maximizing performance while minimizing unnecessary hardware complexity.

2.2.5 Architectural Comparison from a Software Perspective

To further illustrate how NM-Carus differs from simply introducing a regular CPU core near the memory bank, this section compares the implementation of a matrix multiplication operation across three architectures: a conventional RISC CPU, a standard RISC-V vector processor utilizing the RVV extension, and the proposed NM-Carus architecture. The pseudocode provided in Algorithm 1, Algorithms 2 and 3 illustrates the differences in implementation strategies among these systems, reflecting their respective low-level operations. For the CPU and RVV implementations, the pseudocode resembles assembly code to emphasize the instruction-level details, while for NM-Carus, it represents the sequence of operations executed within the device, abstracting away the minimal overhead of offloading the kernel from the host CPU. In the case of a regular CPU core placed near the memory bank, the CPU would still require explicit load and store instructions to move data between the memory and its internal registers. Each computation would involve loading operands into registers, performing arithmetic operations, and storing the results back to memory. This approach not only introduces significant data movement overhead but also requires fetching and executing a large number of instructions, especially for data-intensive operations like matrix multiplication. NM-Carus, on the other hand, operates directly on the data stored in memory without the need for explicit data transfers. Its VPU processes data in place within the VRF, and the use of vector instructions allows a single instruction to specify operations on large datasets. This greatly reduces the number of instructions fetched and executed, and eliminates unnecessary data movement, leading to enhanced energy efficiency and performance.

One of the most significant differences lies in how each architecture handles memory address updates during computation. In the traditional RISC CPU implementation, explicit instructions are required to update memory addresses to traverse the data structures, resulting in a considerable number of additional operations within the inner loops of the algorithm. The RVV vector processor reduces this overhead by employing vector load and store instructions that can handle entire vectors of data, but still relies on explicit address updates for each iteration. In contrast, NM-Carus leverages vector instructions that inherently manage the iteration over data elements, effectively eliminating the need for explicit address manipulation within the inner loops. Moreover, since computations occur directly within the memory, there is no need for load and store instructions to move data to and from processor registers. This not only simplifies the code but also reduces the instruction count and data movement overhead, leading to significant energy savings compared to a regular CPU core near the memory bank. By accessing data directly within the memory hierarchy without traversing complex interconnects and buffers, NM-Carus can more effectively utilize available memory bandwidth

and relax timing constraints. This leads to higher energy efficiency and better performance compared to traditional von Neumann architectures and even conventional vector processors.

Below, the pseudocode for the matrix multiplication implementations on the three architectures is presented, highlighting the differences in their approaches.

RISC CPU Implementation

Algorithm 1 outlines the matrix multiplication routine on a traditional RISC CPU. The algorithm involves three nested loops to traverse the rows and columns of the matrices A and B. Explicit load and store instructions are used to move data between memory and the processor's registers, and address pointers are manually updated after each operation.

Algorithm 1 Matrix multiplication on a conventional RISC CPU (any size, 32-bit)

```
1:  procedure CPUMATMUL
2:      Initialization of pointers and indices
3:      for each row i of A do
4:          for each column j of B do
5:              acc ← 0
6:              for each element k in row i of A do
7:                  x ← LOAD(addr_A)
8:                  y ← LOAD(addr_B)
9:                  acc ← acc + x × y
10:                 addr_A ← NEXTELEM(A)
11:                 addr_B ← NEXTROWELEM(B)
12:             STORE(addr_R, acc)
13:             addr_A ← RESETROWELEM(A)
14:             addr_B ← NEXTCOL(B)
15:             addr_R ← NEXTELEM(R)
16:         addr_A ← NEXTROW(A)
17:         addr_B ← RESETCOL(B)
18:     return
```

Conventional Vector Processor Implementation with RVV

Algorithm 2 presents the matrix multiplication using a conventional vector processor with the RISC-V Vector Extension (RVV). In this approach, vector load and store instructions are employed to handle entire vectors of data, reducing the number of iterations and capitalizing on data-level parallelism. However, explicit address updates are still necessary between iterations, and data must be moved between memory and vector registers.

Algorithm 2 Matrix multiplication using a conventional RVV vector processor (any size, 32-bit)

```
1:  procedure RVVMATMUL
2:      Initialization of pointers and indices
3:      for each row i of A do
4:          x ← LOAD(addr_A)
5:          for each column block j of B do
6:              vy ← VLOAD(addr_B)
7:              vz ← vz + x × vy
8:              addr_B ← NEXTCOLBLOCK(B)
9:          VSTORE(addr_R, vz)
10:         addr_A ← NEXTELEM(A)
11:         addr_R ← NEXTROW(R)
12:         addr_B ← RESETCOL(B)
13:     return
```

NM-Carus Implementation

Algorithm 3 illustrates the matrix multiplication on the proposed NM-Carus architecture. Due to its near-memory computing design, NM-Carus operates directly on the data stored in memory without the need for explicit load and store instructions. The vector instructions in NM-Carus inherently manage data iteration, and address updates are minimized, particularly within the inner loops.

Algorithm 3 Matrix multiplication on NM-Carus (any size, any bitwidth)

```
1:  procedure CARUSMATMUL
2:      Initialization of indices
3:      while not END(R) do
4:          x ← A[A_idx]
5:          A_idx ← A_idx + 1
6:          R[R_idx, :] ← R[R_idx, :] + x × B[B_idx, :]
7:          B_idx ← B_idx + 1
8:          if ENDROW(B) then
9:              B_idx ← 0
10:             R_idx ← R_idx + 1
11:     return
```

Discussion

The comparison among the three implementations highlights several key observations:

- **Instruction Count Reduction:** NM-Carus significantly reduces the total number of instructions executed, as it eliminates the need for explicit load/store operations and

address updates within inner loops. This contrasts with the RISC CPU and RVV implementations, which require additional instructions for data movement and address manipulation.

- **Energy Efficiency:** By minimizing instruction fetches and data transfers, NM-Carus reduces energy consumptiona critical advantage for edge computing applications where power efficiency is paramount.
- **Simplified Code Structure:** The code for NM-Carus is more streamlined, with fewer control flow constructs and a higher level of abstraction in data handling, making it potentially easier to develop and maintain.

While the RVV vector processor improves upon the traditional RISC CPU by leveraging data-level parallelism, it still incurs overhead due to explicit data movement between memory and vector registers. NM-Carus further optimizes this process by integrating computation directly within the memory hierarchy, embodying the principles of near-memory computing.

This architectural approach not only enhances performance but also aligns with the design goals of NMC devicesto provide efficient computation for data-intensive workloads without the penalties associated with data movement in traditional architectures.

2.3 NM-Carus Physical Implementation

To validate the proposed NM-Carus architecture, it was implemented with a memory capacity of 32 KiB using a low-power 65 nm CMOS technology library. The design automation flow utilized Synopsys Design Compiler® 2020.09 for logic synthesis and Cadence Innovus® 20.1 for physical design, including placement and routing.

2.3.1 Design Constraints and Methodology

A primary objective was to create NMC devices that could serve as direct replacements for standard SRAM memories. To achieve this, the cycle time and input/output delay constraints under worst-case conditions were carefully set to match those of a single-port, 32 KiB reference SRAM macro. This reference macro was generated using the same SRAM compiler employed for the internal SRAM banks within the NMC devices. By maintaining consistency in the design constraints and utilizing the same metal stack as standard SRAM macros, we ensured minimal integration effort when incorporating the NMC macros into a host system.

2.3.2 Post-Layout Characteristics

Table 2.3 summarizes the post-layout area and timing characteristics of the NMC macros compared to a traditional SRAM implementation. The results indicate that NM-Carus meets the target clock frequency of 330 MHz, with input and output delays closely matching those of the reference SRAM. Fig. 2.6 presents the post-synthesis area breakdown for NM-Carus highlighting the contributions of various components to the total area.

Table 2.3: Post-layout area and timing characteristics of a 32 KiB traditional SRAM and NM-Carus when implemented on a low-power 65 nm CMOS technology node.

Metric	SRAM	NM-Carus
Post-synthesis Area [μm^2]	–	$315 \cdot 10^3$
Post-layout Area [μm^2]	$200 \cdot 10^3$	$419 \cdot 10^3$ (+110 %)
Max Clock Frequency [MHz]	330	330
Max Input Delay [ns]	0.69	0.70 (+2 %)
Max Output Delay [ns]	2.28	2.48 (+9 %)

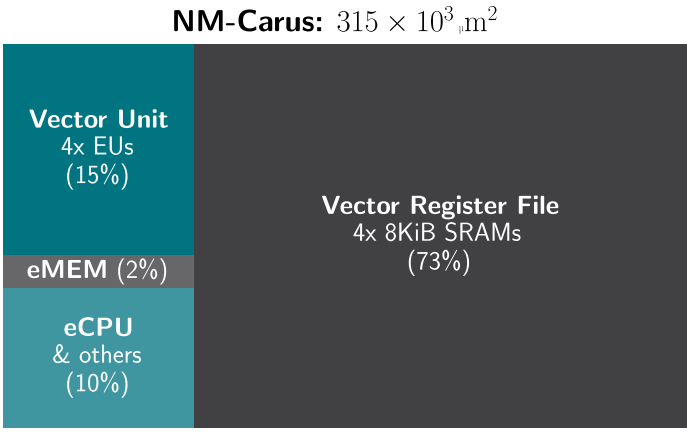


Figure 2.6: Post-synthesis area breakdown of a 32 KiB instance NM-Carus to scale. Data source: [22].

2.3.3 Implementation Details of NM-Carus

The NM-Carus architecture was implemented in a four-lane configuration to balance performance and area constraints. The controller and vector processing unit (VPU) are centrally placed among four 8 KiB SRAM macros that form the internal vector register file (VRF), as depicted in Fig. 2.7. A small embedded memory (eMEM) of 512 B is realized using a register file macro and serves as the code memory for the embedded eCPU.

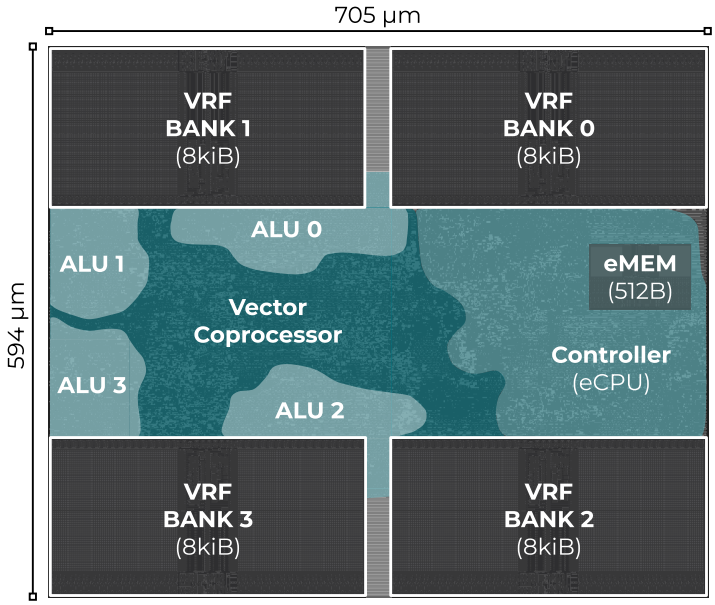


Figure 2.7: Layout of the 32 KiB NM-Carus implementation (4 lanes). Source: [22].

Performance and Timing

The design successfully meets the target clock frequency of 330 MHz. The input and output delays are slightly increased by approximately 2 % and 9 %, respectively, compared to the reference SRAM. This minor increase is attributed to the additional multiplexing logic required to connect either the appropriate VRF bank or the controller bus to the shared external bus interface.

Area Analysis

The post-layout area of NM-Carus measures approximately $705.0 \times 594.4\,\mu m^2$, which is about twice the area of the reference 32 KiB SRAM. This increase aligns with the design goal of achieving a 50% memory to logic area ratio. The area overhead compared to NM-Caesar is

primarily due to the additional logic components in NM-Carus, including the eCPU and the more complex VPU. However, another contributing factor is the sublinear scaling behavior of SRAM footprints relative to their capacity reduction. As illustrated in Fig. 2.6, the four-bank data memory in NM-Carus occupies more area than the two-bank configuration in NM-Caesar, despite both having the same total capacity.

Floorplanning and Routing Considerations

To mitigate routing congestionwhich is a significant challenge due to the increased complexity of the VPU the VRF banks are strategically placed at the corners of the NM-Carus floorplan. This arrangement allows the vector permutation unit, which interconnects all VRF banks, to be centrally located within the integrated circuit (IC), thereby minimizing wire lengths and reducing signal propagation delays.

Additional space is allocated between adjacent memory banks to facilitate the routing of input and output pins from the periphery to the central logic components. This design choice helps limit timing penalties while maintaining compatibility with the pin placement of the reference SRAM macro, where pins are typically located on the bottom side.

Scalability and Technology Node Considerations

Implementations of NM-Carus in more advanced technology nodes could benefit from the higher scaling efficiency of logic circuits compared to SRAM arrays. This would potentially reduce the area overhead associated with the additional logic or allow for a greater number of lanes within the same area budget, thereby enhancing performance without compromising the compactness of the design.

2.4 NM-Carus Performance Assessment

This section evaluates the system-level advantages of integrating the proposed NM-Carus NMC macro into the memory subsystem of a low-power, edge-oriented MCU. The system throughput and energy consumption are assessed over a set of representative computing kernels and compared with their CPU-only counterparts on the same system.

2.4.1 Experimental Setup

System Integration

The 32 KiB NMC macro presented in Section 2.3 was integrated into X-HEEP [49], an open-source, configurable, and low-power RISC-V microcontroller (MCU). One of the eight 32 KiB SRAM banks connected to the X-HEEP bus was replaced by NM-Carus, and the necessary configuration registers were added to the peripheral subsystem of X-HEEP. The NM-Carus interrupt pin was routed to the system CPU, which is the OpenHW Group CV32E40P in-order 4-stage RISC-V processor implementing the RV32IMC extensions. The required input data is directly embedded in the firmware and loaded into the NMC macro at startup to emulate a real-world online data acquisition and processing scenario.

Using worst-case timing constraints, the system was synthesized, placed, and routed using a low-power 65 nm CMOS technology library. All energy data presented in this section was collected from power analysis performed with Synopsys PrimePower® 2019.12 under typical operating conditions, using value change dump (Value Change Dump (VCD)) files generated from system-level post-layout simulations while running the selected benchmark applications. Unnecessary subsystems and interfaces were power-gated during the experiments. The layout of the system is shown in Fig. 2.8.

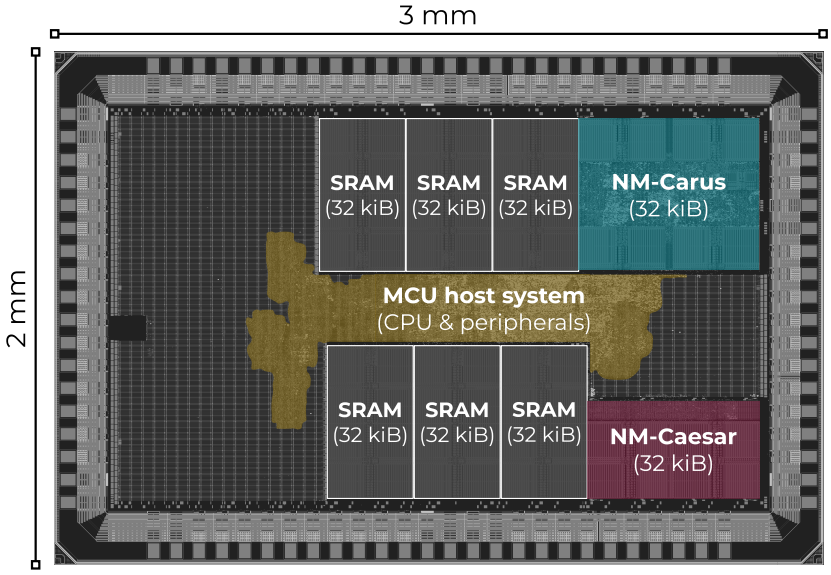


Figure 2.8: Layout of the X-HEEP system enhanced with NM-Carus.

Benchmarks

To assess the system-level execution time and energy efficiency gains enabled by the proposed NM-Carus architecture, a set of representative computing kernels was selected and implemented in both a CPU-only version, used as a baseline, and an NMC-enhanced variant using NM-Carus. The chosen kernels, listed in Table 2.4 and illustrated in Fig. 2.9, constitute the basic building blocks for more complex applications that modern edge devices are expected to run, ranging from simple bitwise or element-wise arithmetic-logic operations to linear algebra (matrix multiplication, 2D convolution) and machine-learning-specific kernels (ReLU, max pooling).

NM-Carus kernels are written in C or assembly, compiled using an extended version of the RISC-V GCC compiler with assembler support for the `xvnmc` vector extension, and copied from memory to the eMEM at runtime. The CPU-only kernels are compiled for the low-power system CPU using the RISC-V RV32IMC instruction set. To ensure a fair comparison, the firmware is always compiled with the `-O2` optimization level.

2.4.2 Experimental Results

Table 2.4 reports the relative execution time and energy reduction of the NMC-enhanced benchmark kernels compared to their reference CPU-only implementations. The speedup in arithmetic and linear algebra kernels ranges from 10.0 × to 55.7 × for NM-Carus, depending on the data width. The execution time is further reduced by up to 99.6 × in the case of the ReLU kernel on NM-Carus, thanks to the inclusion of minimum and maximum selection instructions in its `xvnmc` instruction set, which brings a significant speedup compared to the CPU implementation relying on inefficient data-dependent branches.

On the other hand, the NMC architecture achieves only marginal improvement in the execution time for the max pooling kernel. The lack of vector reduction operations in NM-Carus requires horizontal pooling to be implemented in software on the NM-Carus embedded CPU (eCPU). In general, the energy efficiency of the proposed NMC integrated circuit (Integrated Circuit (IC)) follows the same trend as the throughput, with significant gains over the CPU baseline, as visualized in Fig. 2.9. It is noteworthy that the performance gain when moving to sub-word data sizes is primarily due to the packed-SIMD architecture of the arithmetic units inside NM-Carus. This approach has also been proven effective in conventional computing systems, typically in the form of dedicated instruction set extensions [31].

The data in Table 2.4 was collected during kernel runs over large input datasets to demonstrate the maximum potential of the proposed NM-Carus architecture. To further analyze the performance scaling of NM-Carus, its performance was measured when executing a matrix multiplication kernel on input matrices with varying dimensions. The results of this analysis, reported in Fig. 2.10, highlight how the overhead of the NM-Carus controller affects performance with small workloads. Offloading workloads to NM-Carus incurs an overhead associated with the embedded eCPU executing a sequence of instructions to bootstrap the offloaded kernel. This overhead becomes negligible for larger data sizes, allowing NM-Carus to achieve higher throughput and energy efficiency compared to the CPU-only system, saturating at 0.48 outputspercycle and 66 pJ per output with 8-bit data.

The performance scaling further underscores the importance of architectures like NM-Carus, especially for workloads with large data sizes where the overhead becomes insignificant. This characteristic makes NM-Carus particularly well-suited for data-intensive applications common in edge computing.

The results in Table 2.4 provide another important insight: the energy reduction of the NMC-enhanced kernels is lower than their throughput gain. In other words, the system consumes more power when executing the same workload on the NMC macro compared to the CPU baseline. This phenomenon can be attributed to the increased activity in the memory subsystem and the NM-Carus computational units.

To better analyze the power contributions in the system, a breakdown of the average power consumption during the execution of 8-bit and 32-bit 2D convolution and matrix multiplication kernels is shown in Fig. 2.11. The first observation is that memory accesses represent one of the largest contributors to system power consumption in all cases, highlighting the importance of compute-in-memory (CIM) solutions for enhancing energy efficiency in edge computing. In the CPU case, memory accesses consume approximately as much power as the CPU itself. In the NM-Carus case, the power contribution of its CPU-based controller is negligible compared to the accesses to its four VRF SRAM banks, which account for about 60 % of the total system power. Notably, the computing resources in NM-Carus consume less power than the system CPU while producing significantly more output samples in the same amount of time.

These findings confirm the advantages of exploiting a vector-capable NMC architecture like NM-Carus to enhance performance and energy efficiency in edge devices. By processing data directly within the memory subsystem and leveraging data-level parallelism, NM-Carus reduces the energy overhead associated with data movement and computation, making it an effective solution for energy-constrained applications.

2.4.3 Comparison with the State of the Art

To position the proposed NM-Carus NMC architecture within the context of current Compute-In-Memory (CIM) solutions, it is compared against several recent representative IMC and NMC integrated circuits (ICs): (1) **BLADE** [50], an in-SRAM computing architecture that utilizes local wordline groups for energy-efficient computations; (2) **Computational SRAM (C-SRAM)** [34], a hybrid IMC and NMC scalable integrated vector unit; and (3) **Vecim** [51], a high-performance vector coprocessor based on the ARA architecture [48] that exploits IMC to achieve top-tier energy efficiency.

Table 2.5 provides a quantitative and qualitative overview of the features, architecture, physical characteristics, and performance of these designs compared to NM-Carus. To establish a fair comparison among the different works, the physical characteristics of BLADE and C-SRAM were normalized by applying approximate scaling factors based on the scaling of an SRAM bitcell from their original technologies to the target 65 nm CMOS technology used for NM-Carus and Vecim. Since both designs are memory-dominated, the scaling factors were applied to both the memory and logic areas, resulting in optimistic 65 nm estimates for BLADE and C-SRAM. The energy scaling factors were defined as the ratio between the energy cost of a read operation from a 6T or 8T SRAM of equivalent array size implemented in a 65 nm technology and their original 28 nm (BLADE) or 22 nm (C-SRAM) nodes.

To further compare the performance of these CIM architectures, Table 2.6 reports the peak throughput and energy consumption of BLADE, C-SRAM, and NM-Carus when executing matrix multiplication kernels with different data widths. The throughput of BLADE and C-SRAM was estimated based on the values reported in their respective publications for multiplication operations. Since BLADE and C-SRAM were implemented as 2 KiB and 4 KiB macros, respectively, 16 and 8 instances were considered to match the 32 KiB memory capacity of NM-Carus. A single 32 KiB instance of BLADE is also included to illustrate its scaling with capacity. The estimation does not consider structural hazards or data replication issues for data sizes exceeding the array dimensions in BLADE and C-SRAM. Similarly, when evaluating energy consumption, the proportional increase in static leakage power for a 32 KiB BLADE subarray was considered, neglecting the increase in dynamic power. These assumptions were made to avoid penalizing or underestimating the performance of BLADE and C-SRAM, placing them under optimal conditions for a fair comparison with NM-Carus.

NM-Carus achieves up to 1.97 $\times$ faster execution time than the multi-array BLADE implementation on 16-bit data and performs comparably on 32-bit data. While BLADE outperforms NM-Carus on 8-bit data due to its efficient add-and-shift multiplier architecture for smaller

data types, NM-Carus offers greater versatility and programmability, which may lead to better overall performance in real-world applications where data widths vary. Additionally, NM-Carus is the most energy-efficient design among those compared, consuming up to 3 × less energy than BLADE on 32-bit data. The single-array BLADE instance and the multi-array C-SRAM show lower performance and higher energy consumption compared to NM-Carus and the multi-array BLADE in all cases. It should be noted that the C-SRAM data is derived from silicon measurements, which may account for discrepancies with post-layout simulation results.

Discussion

The comparison reveals that among the considered architectures, NM-Carus achieves the highest peak energy efficiency, being 6 % higher than Vecim and 21 % higher than the multi-array BLADE implementation when normalized to the same technology node. NM-Carus also demonstrates competitive throughput and area efficiency, especially given its fully digital implementation using standard SRAM macros, which simplifies integration and scalability.

Moreover, NM-Carus offers versatility and programmability, supporting a wide range of operations and bitwidths, and requiring minimal deployment constraints compared to other architectures that may necessitate custom SRAM arrays or specific data alignment strategies. This flexibility makes NM-Carus suitable for a variety of applications in edge computing and other energy-constrained environments.

In terms of performance scaling, NM-Carus benefits from its vector processing capabilities, with throughput scaling nearly linearly with the number of lanes. This characteristic allows for performance optimization based on specific application requirements and area or power budgets.

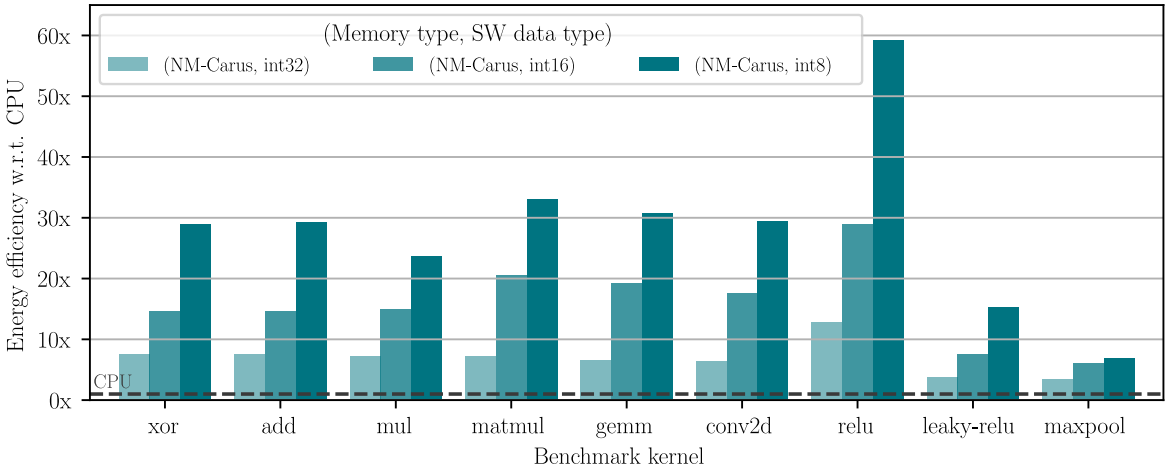
Table 2.4: System-level throughput and energy improvement of the X-HEEP system when executing the same kernels on NM-Carus, relative to their CPU-only implementation (baseline). Improvements are computed as CPU data divided by NM-Carus data (higher is better). Data source: post-layout simulation (65 nm, $f_{CLK} = 250$ MHz).

Device and Bitwidth		Bitwise XOR[a]		Element-wise Addition[a]		Element-wise Multiplication[a]		Matrix Multiplication[b]		GEMM[c]	
Baseline		**Cycles/output**	**Energy/output**	**Cycles/output**	**Energy/output**	**Cycles/output**	**Energy/output**	**Cycles/output**	**Energy/output**	**Cycles/output**	**Energy/output**
RISC-V CPU (RV32IMC)	8-bit	11.0	270 pJ	11.0	271 pJ	11.0	267 pJ	104.0	2.67 nJ	108.1	2.83 nJ
	16-bit	11.0	273 pJ	11.0	270 pJ	11.0	285 pJ	106.0	2.85 nJ	108.2	3.00 nJ
	32-bit	10.0	280 pJ	10.0	278 pJ	10.0	279 pJ	90.1	2.57 nJ	90.3	2.65 nJ
Improvement		**Throughput**	**Energy**	**Throughput**	**Energy**	**Throughput**	**Energy**	**Throughput**	**Energy**	**Throughput**	**Energy**
NM-Carus (32 KiB, 4 lanes)	8-bit	55.7×	28.9×	55.7×	29.2×	42.0×	23.7×	50.0×	33.1×	46.7×	30.7×
	16-bit	27.9×	14.7×	27.9×	14.7×	27.9×	14.9×	35.1×	20.6×	32.5×	19.2×
	32-bit	12.7×	7.5×	12.7×	7.5×	12.6×	7.1×	11.2×	7.2×	10.0×	6.5×

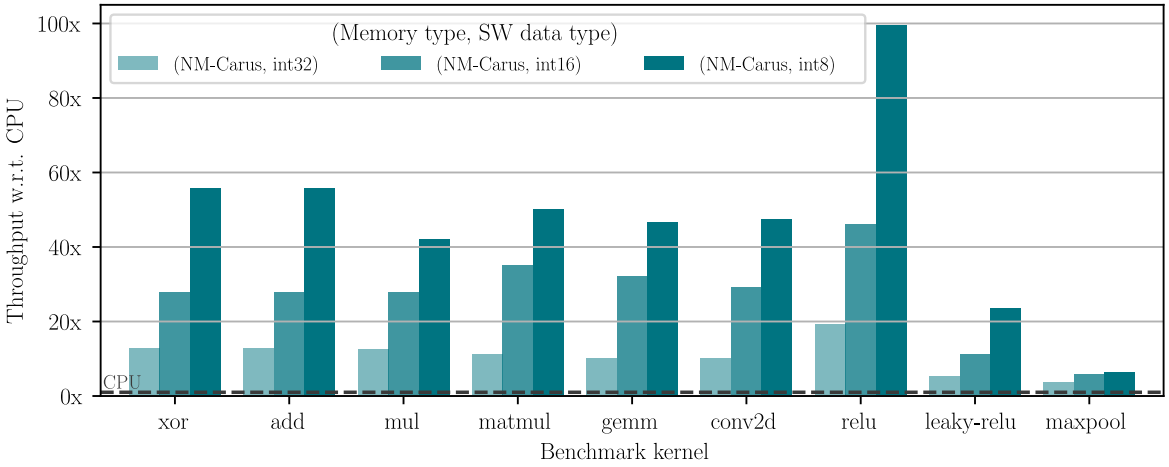
[a] Input data size: 10 KiB (CPU and NM-Carus). [b] $A[8,8] \times B[8,p]$, with $p = \{256, 512, 1024\}$ (CPU and NM-Carus) for $\{32, 16, 8\}$ bits.
[c] $\alpha(A[8,8] \times B[8,p]) + \beta C[8,p]$, with $p = \{256, 512, 1024\}$ (CPU and NM-Carus) for $\{32, 16, 8\}$ bits.

Device and Bitwidth		2D Convolution[d]		ReLU[e]		Leaky ReLU[e,f]		Max Pooling[g]	
Baseline		**Cycles/output**	**Energy/output**	**Cycles/output**	**Energy/output**	**Cycles/output**	**Energy/output**	**Cycles/output**	**Energy/output**
RISC-V CPU (RV32IMC)	8-bit	135.0	3.3 nJ	13.0	340 pJ	10.5	258 pJ	66.3	1.46 nJ
	16-bit	133.0	3.4 nJ	12.0	335 pJ	10.0	256 pJ	67.4	1.50 nJ
	32-bit	115.1	3.1 nJ	10.0	299 pJ	9.5	256 pJ	48.2	1.10 nJ
Improvement		**Throughput**	**Energy**	**Throughput**	**Energy**	**Throughput**	**Energy**	**Throughput**	**Energy**
NM-Carus (32 KiB, 4 lanes)	8-bit	47.5×	29.4×	99.6×	59.2×	23.5×	15.2×	6.4×	6.9×
	16-bit	29.3×	17.6×	46.0×	30.7×	11.2×	7.5×	5.9×	6.0×
	32-bit	10.1×	6.3×	19.2×	12.8×	5.3×	3.8×	3.5×	3.3×

[d] $A[8,n] \circledast F[f,f]$, with $n = \{256, 512, 1024\}$, $f = 3$ (CPU and NM-Carus) for $\{32, 16, 8\}$ bits. [e] Input data size: 16 KiB (CPU and NM-Carus). [f] Negative slope coefficient implemented as right shift (only powers of 2). [g] Pooling window: $W[2,2]$ (stride: 2). Data size: 16 KiB (CPU and NM-Carus).

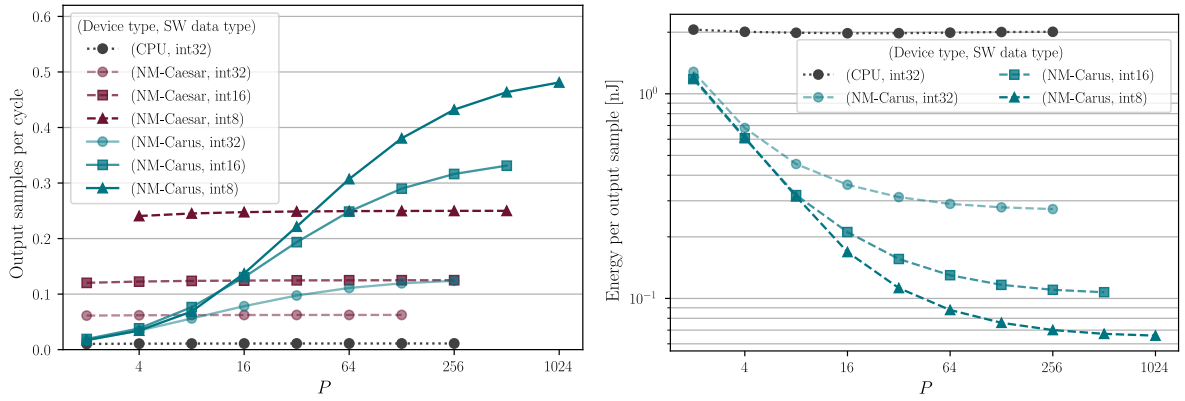


(a) Energy efficiency improvement.



(b) Throughput improvement.

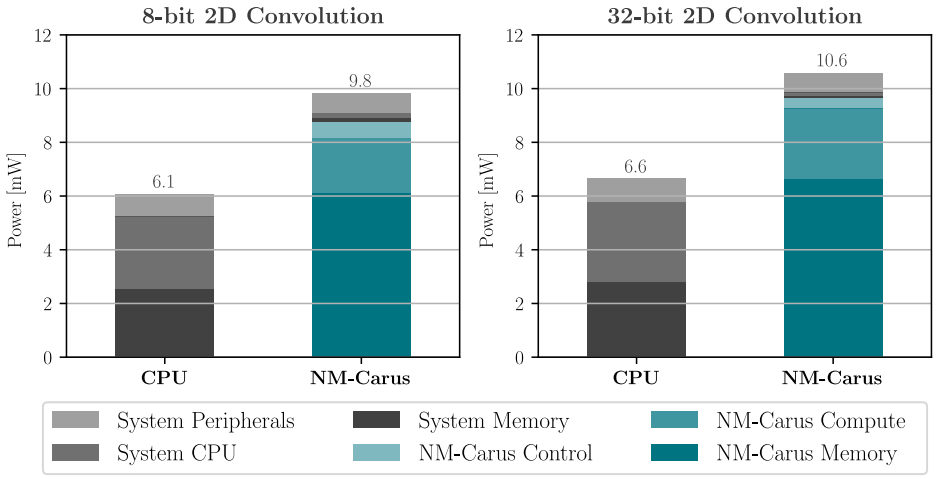
Figure 2.9: Energy efficiency gain (a) and throughput improvement (b) of the NMC-enhanced X-HEEP MCU compared with its CPU-only version. Data source: post-layout simulation (65 nm, $f_{\text{CLK}} = 250\,\text{MHz}$).



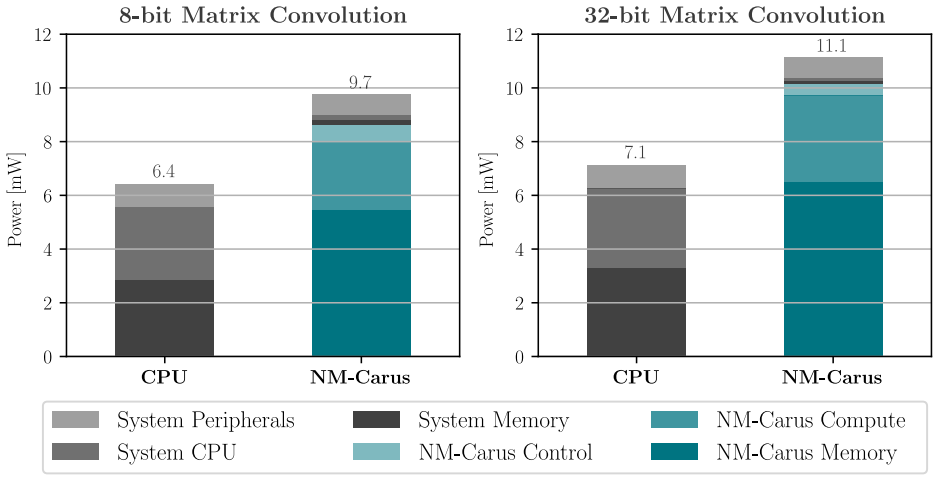
(a) Matrix mult. ($[8,8] \times [8,P]$) throughput scaling.

(b) Matrix mult. ($[8,8] \times [8,P]$) energy scaling.

Figure 2.10: Throughput (a) and energy (b) scaling of NM-Carus (cyan) compared to the CPU-only MCU version (grey) on several matrix multiplication kernels with different shapes and bitwidths. The CPU throughput does not vary significantly with data size when using the standard RV32IMC ISA; therefore, only 32-bit data is shown. Data source: post-layout simulation.



(a) 2D convolution.



(b) Matrix multiplication.

Figure 2.11: Average power breakdown in CPU (grey) and NM-Carus (cyan) for 8-bit and 32-bit 2D convolution (a) and matrix multiplication (b). Data source: post-layout simulation.

Table 2.5: Comparison with existing state-of-the-art IMC and NMC solutions.

	BLADE [50]		**C-SRAM** [34], [52]		**Vecim** [51]	**NM-Carus**
CIM type	IMC		IMC + NMC		IMC + NMC	NMC
Array instances	16 × 2 KiB		4 × 8 KiB		1 × 16 KiB (4 lanes)	1 × 32 KiB (4 lanes)
SRAM type	Foundry 6T, custom array		Foundry 8T, custom array		Foundry 8T, custom array	Foundry 6T
Useful bitcell density[a] [%]	**53.5**		20.3		1.7	33.0
Deployment constraints	• Word alignment • Data placement (LG)		• Word alignment • Data replication ($D, \overline{D}$)		• Vector alignment	• Vector alignment
Instance ALU features	32-bit SIMD ALU		128-bit SIMD ALU		4 × 256-bit SIMD ALUs	4 × 32-bit SIMD ALUs
Multiplier architecture	1-bit add and shift		Add and shift		Double-rate bit-parallel	1 × 16-bit mult. per ALU
Supported operations	• ADD/SUB • MULT • Logic		• ADD/SUB • MULT • MAC • Logic • Copy • Shuffle • Comparison • Logical & arith. shift		• RVV instructions (integer and floating-point)	• RV32EC ISA • ADD/SUB • MULT • MAC • Comparison • Logical & arith. shift • Logic • Copy • Slide
Technology	28 nm	65 nm[b]	22 nm	65 nm[b]	65 nm	65 nm
Area [μm^2]	$64 \cdot 10^3$	$580 \cdot 10^3$	$17.5 \cdot 10^3$	N/A[c]	$4 \cdot 10^6$	$415 \cdot 10^3$
Nominal frequency [MHz]	2200	330[d]	1000	330[d]	250	330
Supported bitwidths	32, 16, 8	32, 16, 8	64, 32, 16, 8	64, 32, 16, 8	32, 16, 8	**32, 16, 8**
Peak throughput[e] [GOPS]	35.2	5.3	10.7	3.5	31.8	2.64
Energy efficiency [GOPS/W]	830.7[f]	254.2[f]	52.0	13.2	289.1	**306.7**[g]
Area efficiency [GOPS/mm^2]	550[f]	9.1	611	N/A[c]	8.0	6.4

[a] Area of useful bitcells (i.e., excluding replicated data), normalized to 6T bitcell area.
[b] Area, timing, and power at 65 nm were estimated from 28 nm data based on commercial SRAM scaling for consistency.
[c] Area estimation from 22 nm data is not straightforward due to C-SRAM's mixed IMC/NMC design; therefore, it is not provided.
[d] Assumed to match the operating frequency of the 65 nm 32 KiB SRAM used as a reference for NM-Carus timing constraints. [e] 8-bit MAC operations unless specified otherwise. One MAC operation is considered as two elementary operations (one multiplication and one addition).
[f] Without controller overhead. [g] Average power from post-layout simulation while running a matrix multiplication kernel.

Table 2.6: Peak performance comparison on matrix multiplication kernels.

Metric and Bitwidth		**BLADE**[a,b] 16×2 KiB		**BLADE** 1×32 KiB		**C-SRAM**[a,b] 8×4 KiB	
		28 nm	65 nm	28 nm	65 nm	22 nm	65 nm
Cycle count	8-bit[d]	**12.8k**		204.8k		19.2k	
	16-bit[e]	25.6k		409.6k		38.4k	
	32-bit[f]	51.2k		819.2k		76.8k	
Execution time[c] [ţs]	8-bit[d]	5.8	**38.8**	93	620	19.3	58.1
	16-bit[e]	11.6	77.5	186	1240	38.4	116
	32-bit[f]	23.3	155	372	2480	76.8	232
Energy [pJ/MAC]	8-bit[d]	2.4	7.9	13	43	38.8	150
	16-bit[e]	8.1	26.7	29.4	97.1	155	600
	32-bit[f]	31.1	103	96.9	320	621	2400

[a] Data replication latency is neglected (best-case scenario). [b] Inter-bank data placement hazards (e.g., MAC source operands in different banks) are neglected (best-case scenario).
[c] At nominal frequency. [d] $A[10,10] \times B[10,1024]$ [e] $A[10,10] \times B[10,512]$
[f] $A[10,10] \times B[10,256]$

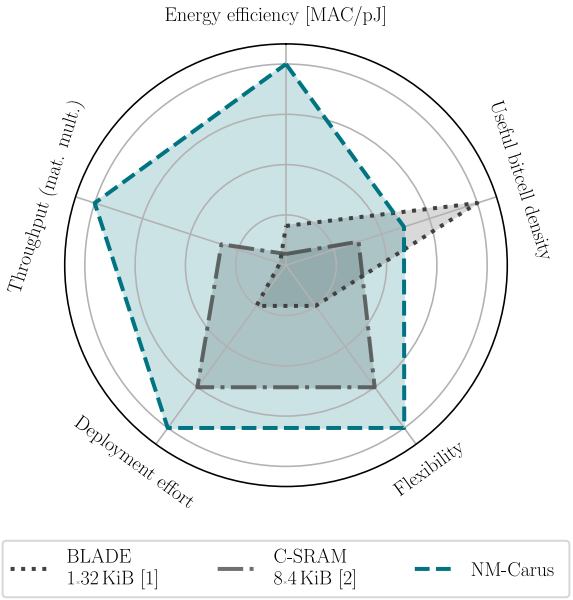


Figure 2.12: Comparison of the normalized performance and characteristics of NM-Carus with state-of-the-art architectures. *Energy efficiency*, *Throughput*, and *Useful bitcell density* are quantitative metrics taken from Tables 2.5 and 2.6. The *Versatility* score considers qualitative aspects such as ease of use (e.g., programmability in the case of NM-Carus), the range of supported operations (including special-purpose ones like shuffle operations in C-SRAM [52]), and data placement or replication constraints. *Deployment effort* refers to the complexity of the ASIC deployment process and the effort required to port the device to a different technology node. Fully digital NMC architectures using commercial off-the-shelf SRAM are therefore assigned a higher score.

Chapter 3

LEN5: Out-of-Order CPU Design for Edge Computing

3.1 Limitations of In-Order CPUs in Edge Computing

The advent of data-driven computing paradigms has significantly increased the demand for enhanced processing capabilities and energy efficiency in edge-oriented systems-on-chip (SoCs). Applications such as health monitoring [53], robotics [54], and autonomous driving [55] increasingly rely on advanced functionalities provided by artificial neural networks (ANNs), which are being embedded into Internet of Things (IoT) devices. However, traditional von Neumann architectures and in-order central processing units (CPUs) face inherent limitations in meeting these growing demands, particularly due to bandwidth and latency constraints in centralized computing infrastructures.

To address these challenges, heterogeneous embedded SoCs have emerged as a promising solution. These systems offload computationally intensive tasks to specialized, tightly-coupled coprocessors or accelerators that implement domain-specific instruction set architecture (ISA) extensions. For instance, multiply-accumulate operations, which form the core of ANN workloads and are replicated extensively, are ideal candidates for acceleration in execution units (Execution Units (EUs)). Low-power microcontrollers (MCUs) equipped with single instruction, multiple data (SIMD) or vectorized datapath units (Dot-Product Units (DPUs)) embedded within the CPU pipeline [56], [57], or with specialized coprocessors [58], are becoming prevalent in various application fields.

While these accelerators offer superior execution speed and energy efficiency compared to the main CPU, integrating them into systems presents significant challenges. In traditional in-order CPUs, offloaded instructions involving computationally intensive operations can result in long and variable latencies, ranging from tens to thousands of cycles. When the host CPU waits for the coprocessor to produce results, it may stall, leading to underutilization of both the CPU and the accelerator's potential performance benefits.

Optimizing coprocessor utilization while mitigating stalls on the host CPU is a complex task that cannot be fully resolved at compile time, especially in scenarios where the same ISA extension may be implemented differently across various microarchitectures. In-order CPUs execute instructions sequentially, and any long-latency operation can cause the entire pipeline to stall until the operation completes. This limitation becomes more pronounced with the increasing complexity and variability of workloads in modern embedded systems.

Moreover, programmability is a crucial requirement in many IoT applications, where algorithms must adapt to dynamic input data or environmental changes. For example, online training of ANNs requires high numerical precision during backpropagation, further exacerbating latency issues when using in-order CPUs. Support for floating-point arithmetic can negatively impact overall execution time due to the extended latencies involved.

Another key aspect when deploying edge-computing devices is data security. The RISC-V cryptography vector extension [59] accelerates mathematical operations commonly used in encryption and decryption algorithms. Integer and floating-point polynomial multiplication are examples of data-intensive operations supported by this new ISA extension. The increased complexity of post-quantum cryptographic algorithms [60] further emphasizes the need for efficient acceleration, which can strain the capabilities of in-order CPUs.

The limitations of in-order CPUs in handling long-latency operations and efficiently utilizing tightly-coupled coprocessors present significant challenges in modern edge computing systems. Exploiting instruction-level parallelism (ILP) in addition to data-level parallelism (Data-Level Parallelism (DLP)) is becoming crucial to overcome these hurdles. By reordering instructions and executing independent operations in parallel, the CPU can hide the latency of long-running instructions, reducing stalls and improving overall system performance.

One approach is to adopt strategies from general-purpose, application-class microprocessors that leverage out-of-order (OoO) execution. OoO execution is a well-established technique for optimizing instructions per cycle (IPC) by allowing the CPU to execute instructions based on operand availability rather than their original program order. This dynamic scheduling enables the CPU to continue processing independent instructions while waiting for the results of long-latency operations, effectively masking their delay.

However, implementing OoO execution and dynamic scheduling in embedded systems introduces increased complexity, area overhead, and energy consumption [61]. The control logic required for dependency checking, instruction reordering, and maintaining correct program semantics becomes more intricate. Therefore, balancing the trade-off between performance improvement and efficiency degradation is essential. Modularity and configurability are key features in designing processors that can adapt to varying application needs without incurring prohibitive costs.

In summary, the inherent limitations of in-order CPUs in dealing with long-latency instructions and the growing reliance on tightly-coupled coprocessors necessitate advanced microarchitectural solutions. Exploring techniques such as OoO execution and dynamic scheduling is critical to fully leverage the capabilities of specialized accelerators in heterogeneous embedded SoCs, thereby meeting the increasing performance and efficiency demands of modern edge computing applications.

3.2 Proposed Out-of-Order Microarchitecture

To overcome the limitations of in-order CPUs in handling long-latency instructions and efficiently utilizing tightly-coupled coprocessors, we propose an out-of-order (OoO) microarchitecture designed to maximize instruction-level parallelism (ILP) and mask instruction latencies. The proposed architecture is exemplified by the LEN5 microprocessor, a highly configurable and modular 64-bit RISC-V core featuring speculative OoO execution and OoO commit capabilities. The main objectives of this microarchitecture are:

(a) Provide a scalable and extensible core infrastructure that can be easily tailored to diverse computing requirements.

(b) Effectively hide the latency and manage dependencies of executed instructions to maintain high performance, even when integrating serial or heavily pipelined execution units (EUs).

Figure 3.1 presents the top-level block diagram of the LEN5 microprocessor, highlighting its key components and data flow.

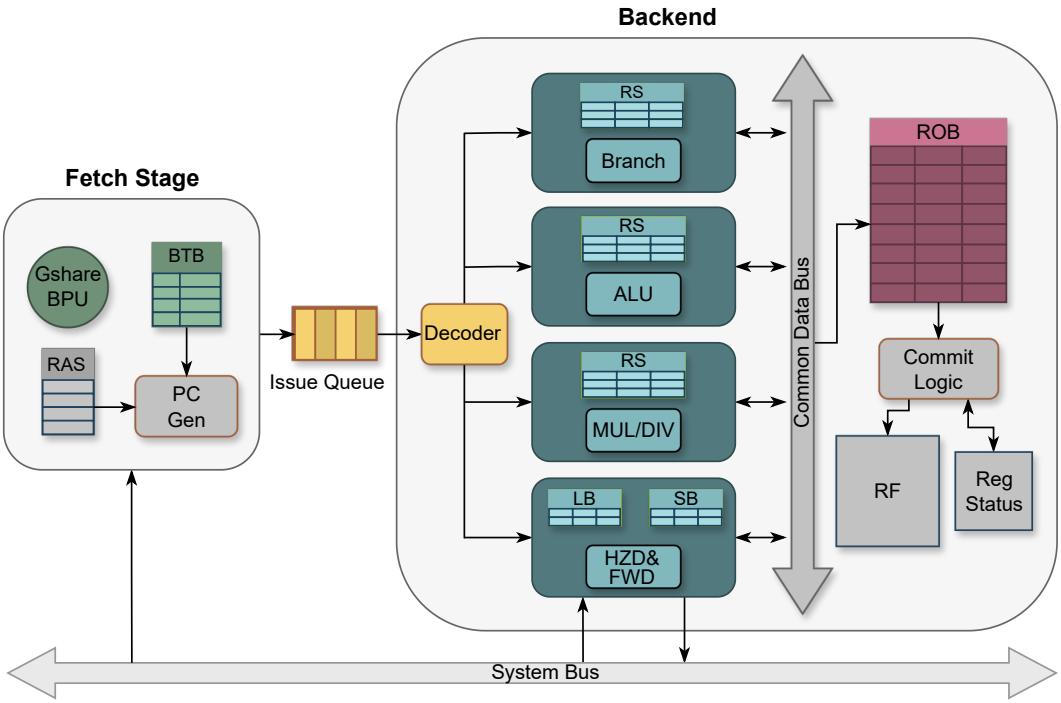


Figure 3.1: Block diagram of the LEN5 microprocessor. Source: [24].

3.2.1 Instruction Fetch and Branch Prediction

The instruction fetch unit is responsible for supplying a steady stream of instructions to the pipeline, aiming to keep the execution units busy at all times. To mitigate the control hazards associated with branch instructions, LEN5 employs a speculative frontend featuring a *gshare* branch predictor, a branch target buffer (BTB), and a return address stack (RAS), all of which are configurable in size. This combination allows the processor to predict the outcome of branch instructions and speculatively execute along the predicted path, thereby reducing stalls caused by control dependencies.

When a branch or jump instruction is fetched, the branch prediction unit (BPU) speculatively updates the program counter (PC) based on the predicted target address. An early decode unit recognizes known static jumps (`JAL`) and subroutine returns (`RET`), utilizing the RAS to update the PC accordingly. This speculative execution continues until the actual outcome of the branch is resolved in the execution stage. In case of a misprediction, the frontend is promptly notified, allowing it to fetch the correct instructions while the backend flushes the mispredicted instructions from the pipeline.

Instructions fetched are placed into an instruction queue (IQ), where they await decoding and dispatch to the execution units. The use of an IQ decouples the fetch and decode stages,

providing additional buffering that helps absorb fluctuations in instruction supply and demand, especially during pipeline stalls or cache misses.

3.2.2 Out-of-Order Instruction Execution

The core of the microarchitecture leverages an enhanced version of Tomasulo's algorithm [62] for dynamic scheduling and OoO execution. Instructions are decoded in program order but can be executed out of order, based on operand availability and resource readiness. This approach allows the processor to exploit ILP by identifying and executing instructions that are independent of prior instructions, thus masking the latency of long-running operations.

Upon decoding, instructions are dispatched to their corresponding reservation stations (RSs), which act as buffers holding instructions until all their operands are ready. Simultaneously, each instruction is assigned an entry in the reorder buffer (ROB), which tracks the status of in-flight instructions and ensures correct commit order. Operand values are sourced either from the physical register file (RF), the ROB, or forwarded via the common data bus (CDB) if produced by earlier instructions that have completed execution.

If an operand is not immediately available, the instruction remains in the RS and monitors the CDB for the required data. Once all operands are ready, the instruction becomes eligible for execution. The execution units can then select ready instructions from their respective RSs, potentially executing multiple instructions in parallel if resources permit.

The CDB plays a crucial role in forwarding results to dependent instructions and the ROB. When an instruction completes execution, it broadcasts its result and destination register tag on the CDB. Instructions in the RSs listening for this tag can capture the result, updating their operands and becoming ready for execution. This mechanism efficiently resolves data hazards without the need for a centralized instruction status table.

The distributed nature of the RSs and the modular design of the execution units facilitate scalability and ease of integration for custom extensions or specialized coprocessors. Adding new execution units involves extending the main instruction decoder, introducing dedicated RSs, and connecting the new functional units to the backend, all while leveraging the existing dependency resolution and forwarding infrastructure.

3.2.3 Load and Store Handling

Memory operations introduce additional challenges due to potential data hazards and variable memory access latencies. The load-store unit (LSU) in LEN5 addresses these issues by implementing an out-of-order load buffer (LB) and an in-order store buffer (SB). The LB allows multiple outstanding load requests, handling out-of-order memory responses and enabling the execution of independent instructions while waiting for memory accesses to complete.

The SB not only queues store instructions but also acts as a level-zero cache, temporarily holding committed store data. This design enables store-to-load forwarding, where subsequent load instructions can directly obtain data from the SB if they access the same memory address, thus bypassing the memory hierarchy and reducing load latency [63].

3.2.4 Out-of-Order Instruction Commit

After execution, instructions are candidates for retirement, where they update the architectural state of the processor. The ROB ensures that instructions commit in a manner that preserves program order and correctness, even though they may have executed out of order. LEN5 employs an OoO commit mechanism with two commit slots:

1. **In-order commit slot**: Always points to the oldest instruction in the ROB and ensures that instructions affecting control flow (e.g., branches, exceptions) are committed in order, preserving program semantics.
2. **Out-of-order commit slot**: Allows newer instructions that have completed execution to commit before older ones, provided certain conditions are met:
    - The instruction has completed execution and its result is available.
    - There are no unresolved branch mispredictions or exceptions preceding it.
    - Committing the instruction does not cause write-after-write (WAW) hazards; that is, no younger instruction writes to the same destination register.

This dual-slot commit strategy helps free up entries in the ROB more quickly, reducing backpressure on the issue stage and allowing more instructions to be in flight. However, the current implementation of LEN5 still allocates ROB entries in order, which can limit the ability to issue new instructions when the ROB is full, particularly in the presence of very long-latency instructions. Addressing this limitation by enabling out-of-order ROB allocation is an area of ongoing development.

3.2.5 Latency Hiding and Dependency Resolution

The effectiveness of the proposed microarchitecture in hiding instruction latency hinges on several key factors:

1. **Instruction window size**: A larger pool of in-flight instructions increases the likelihood of finding independent instructions that can be executed while waiting for long-latency operations to complete.
2. **Execution unit availability**: Multiple execution units and sufficient resources allow parallel execution of independent instructions, maximizing throughput.
3. **Efficient dependency tracking**: The use of tags, RSs, and the CDB enables precise tracking and resolution of data hazards, allowing instructions to execute as soon as their operands are ready.

Figure 3.2 illustrates the backend architecture of LEN5 and provides an example of out-of-order execution and commit. In this scenario, a long-latency coprocessor instruction is in flight, and dependent instructions are waiting for its result. Meanwhile, independent instructions have been executed and are eligible for out-of-order commit.

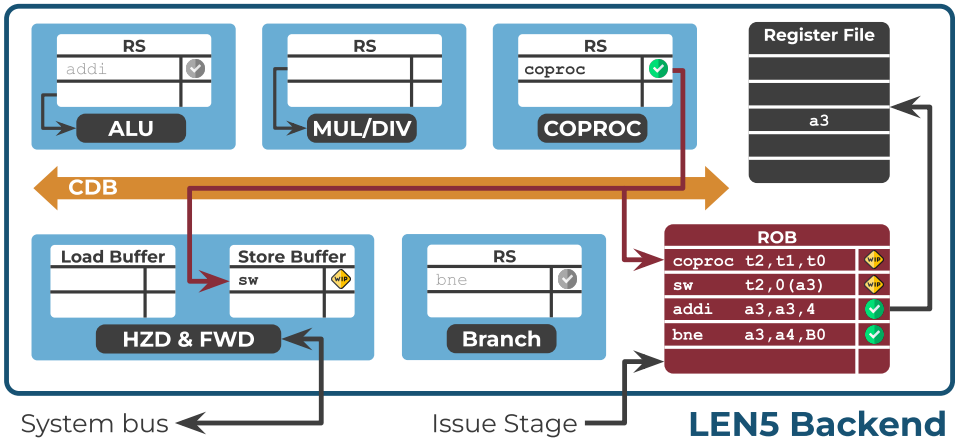


Figure 3.2: Backend architecture of LEN5 with an example of out-of-order execution and commit. The `sw` instruction is waiting for the result (`t2`) produced by the long-latency `coproc` instruction (RAW hazard). Independent instructions (`addi`, `bne`) have completed execution out of order and are eligible for commit. Source: [23].

3.2.6 Scalability and Modularity

The microarchitecture's design emphasizes scalability and modularity, allowing for easy extension and customization to meet specific application requirements. The use of a system-wide *valid-ready* handshake protocol enables each component to operate independently, accommodating varying latencies without introducing global stalls. Internal buffers and queues provide elasticity, absorbing timing variations and decoupling pipeline stages.

Adding support for custom ISA extensions or specialized coprocessors involves minimal modifications:

1. Extending the instruction decoder to recognize new instructions and determine their operand requirements.
2. Introducing dedicated RSs for the new execution units, connected to the existing dependency tracking and scheduling infrastructure.
3. Integrating the new execution units, which adhere to the established *valid-ready* interface for inputs and outputs.

This approach simplifies the integration of domain-specific accelerators, such as vector units or cryptographic engines, enhancing the processor's versatility without significantly impacting timing performance or area overhead.

3.2.7 Implementation Considerations

While the proposed microarchitecture offers significant performance benefits by exploiting ILP and masking instruction latency, it also introduces additional complexity in control logic and resource management. The increased area and power consumption associated with the ROB, RSs, and dependency tracking mechanisms must be carefully balanced against the performance gains, especially in energy-constrained embedded systems.

Configurability is a key aspect of the design, allowing system architects to adjust parameters such as the sizes of the ROB, RSs, branch predictor tables, and execution unit resources to align with specific performance targets and physical constraints. For example, reducing the ROB size may save area and power but could limit the processor's ability to hide long latencies, affecting throughput.

3.3 LEN5 Physical Implementation

LEN5 was synthesized using Synopsys Design Compiler with the TSMC 65 nm low-power (LP) CMOS technology library, suitable for energy-constrained edge devices, under worst-case operating conditions. To showcase the modularity and configurability of LEN5, three core configurations targeting different deployment scenarios were defined, each with varying area and performance requirements:

1. **Max Performance**: This configuration supports the RV64IM RISC-V ISA with a 2-stage pipelined 64-bit multiplier and a serial divider. It includes larger data structures, such as a 32-entry reorder buffer (ROB), 8-entry reservation stations (RSs), and a 16-entry store buffer (SB), to minimize structural hazards and support up to 32 in-flight instructions.

2. **Average Performance**: An intermediate configuration without a hardware divider and with a 32-bit multiplier executing 64-bit operations in 3–4 cycles. It balances performance and area by reducing the size of certain data structures.

3. **Minimal Area**: A configuration optimized for minimal area, omitting both the hardware multiplier and divider, suitable for applications that can tolerate longer execution times.

Table 3.1 summarizes the area and clock frequency of these configurations, along with a comparison to the 32-bit in-order CV32E40P core [64] and the application-class 64-bit in-order CVA6 core [61]. The **Max Performance** configuration of LEN5 has approximately 1.6× the area of the **Average Performance** variant and 2.0× that of the **Minimal Area** variant. Despite the increased area compared to the CV32E40P, LEN5 achieves higher clock frequencies due to its optimized pipeline and sampling of incoming data before forwarding or decoding. Moreover, the largest LEN5 core configuration has a comparable area to the 64-bit in-order CVA6 core, despite implementing an out-of-order execution pipeline. The table also reports the system-level area overhead when replacing the CV32E40P with LEN5 and CVA6 inside the open-source X-HEEP MCU platform [49], in its default configuration with 256 KiB of on-chip SRAM memory. This configuration reflects a common edge-oriented, low-power MCU system with no coprocessors or accelerators, thus representing a worst-case deployment scenario for larger cores like LEN5 and CVA6 in terms of area overhead. As shown, LEN5 introduces a modest area increase of 5.6 % to 11.3 % compared to the same system equipped with the CV32E40P. Depending on the target deployment scenario, application requirements, and energy budget, this area overhead can be considered acceptable given

the performance benefits brought by the increased operating frequency and, most importantly, the IPC improvements in cases where the CPU is coupled with long-latency coprocessors, as demonstrated in Section 3.4.

Table 3.1: Post-synthesis area and clock frequency comparison of LEN5 configurations, CVA6, and CV32E40P when implemented in a low-power 65 nm CMOS technology.

Configuration	**Clock Freq** [MHz]	**Area** [μm^2]	**Area** [kGE][a]	**Relative Area** vs. CV32E40P	**Area Overhead** in X-HEEP[b] [49]
LEN5 Max Perf	438	$423 \cdot 10^3$	294	7.5 ×	+11.3 %
LEN5 Balanced	406	$251 \cdot 10^3$	174	4.4 ×	+7.3 %
LEN5 Min Area	448	$205 \cdot 10^3$	142	3.6 ×	+5.6 %
CVA6 [61]	1700[c]	$368 \cdot 10^3$[d]	210	5.4 ×	+11.1 %
CV32E40P [64]	360	$56 \cdot 10^3$	39	1.0 ×	+0 %

[a] GE: Gate Equivalents, with 1 GE defined as the area of a two-input NAND gate with unitary drive strength. [b] Configured with the default embedded peripherals and 256 KiB of on-chip SRAM memory. [c] When implemented in 22 nm FD-SOI technology.
[d] 65 nm area estimated based on the area in kGE reported for its 22 nm implementation in [61].

Figure 3.3 illustrates the area breakdown of the **Max Performance** configuration. The 64-bit pipelined multiplier accounts for the largest portion of the execution units' area, followed by the arithmetic logic unit (ALU) due to its 8-entry RS. The serial divider contributes approximately 7 % of the execution stage area.

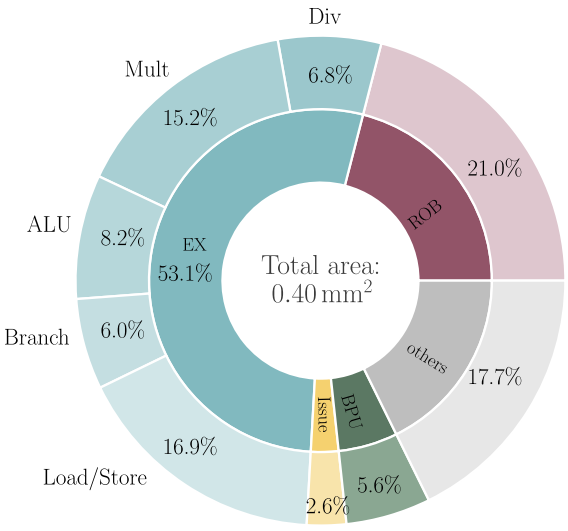


Figure 3.3: Area breakdown of the LEN5 **Max Performance** configuration.

To assess the impact of integrating LEN5 into a typical microcontroller unit (MCU) system, the X-HEEP MCU was used as host platform. X-HEEP was configured with eight 32 KiB SRAM banks and the CV32E40P core. Replacing CV32E40P with LEN5 **Max Performance** results in an overall area increase of approximately 13 %, which is acceptable given the performance benefits discussed in the following sections.

3.4 Performance Assessment

In this section, the performance of the proposed LEN5 microarchitecture are evaluated, focusing on its ability to exploit instruction-level parallelism (ILP) to hide the latency of long-latency instructions, particularly when integrated with tightly coupled coprocessors. LEN5 is compared against the in-order CV32E40X core [31], using the X-HEEP microcontroller [49] as a common platform for fair comparison. The assessment includes area and timing characteristics from logic synthesis and detailed IPC analysis using cycle-accurate RTL simulations with synthetic benchmarks.

3.4.1 Synthetic Benchmarking with Configurable Coprocessor

To evaluate the ability of LEN5 to hide the latency of long-latency instructions, a synthetic benchmarking framework was designed. It consists of a configurable latency coprocessor (CLC) and an automatic code generator that produces test applications with varying instruction compositions and dependency patterns.

Configurable Latency Coprocessor (CLC)

The CLC emulates the behavior of coprocessors with arbitrary latency in both iterative (e.g., vector processors) and pipelined (e.g., data-flow accelerators) modes. Its latency and operating mode can be configured at runtime using custom RISC-V instructions:

```
xdummy.iter rd, rs1, imm # Iterative mode, imm = latency
xdummy.pipe rd, rs1, imm # Pipelined mode, imm = pipeline depth
```

The CLC is integrated into LEN5 by adding a dedicated 4-entry RS and extending the main instruction decoder. The coprocessor uses a *valid-ready* handshake protocol for input

and output transactions, allowing seamless communication with the CPU pipeline. For comparison, the CLC is also integrated into the CV32E40X core using the CORE-V X-Interface (XIF) [47].

Automatic Code Generator

The automatic code generator produces test applications by populating a loop with a CLC instruction followed by a configurable number of independent basic instructions (NIB), such as arithmetic and logic operations from the base RISC-V ISA. The generator allows us to explore various scenarios by adjusting the coprocessor latency, the number of basic instructions, and the presence of data dependencies between loop iterations.

Examples of generated code include:

```
loop:
  xdummy.iter a0, a1, 5
  add         a2, a4, a3
  xor         a2, a5, a4
  addiw       a5, a5, -1
  bnez        a5, loop
```

(a) Independent iterations.

```
loop:
  xdummy.iter a0, a0, 5
  add         a2, a4, a5
  xor         a2, a4, a5
  addiw       a3, a3, -1
  bnez        a3, loop
```

(b) Dependent iterations.

```
loop:
  xdummy.iter a0, a0, 5
  jal         bookkeeping
  add         a0, a0, s1
  addiw       s0, s0, -1
  bnez        s0, loop
```

(c) Dependent iterations with housekeeping.

Listing 3.1: Examples of generated assembly code for testing.

3.4.2 Instruction Per Cycle (IPC) Analysis

A series of experiments to compare the IPC performance of LEN5 and CV32E40X when executing the generated test applications was conducted. The tests varied the coprocessor latency from 1 to 20 cycles and the number of basic instructions (NIB) from 1 to 20. Both cores were simulated using Verilator, and the code was compiled with GCC using the `-O2` optimization level.

Independent Loop Iterations

In the first set of experiments, scenarios where loop iterations are independent were considered, and the CLC instructions do not have data dependencies between iterations (List-

ing 3.1a). Listing 3.4 shows the IPC results for both iterative and pipelined modes of the CLC.

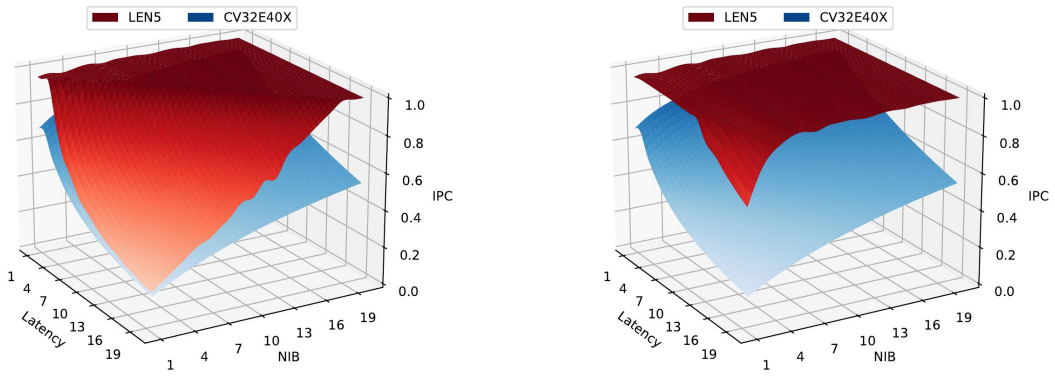


Figure 3.4: IPC comparison without housekeeping and without dependencies using iterative (left) and pipelined (right) CLC. Source: [23].

The in-order CV32E40X core experiences significant performance degradation as the coprocessor latency increases, with minimal improvement from increasing the NIB. In contrast, LEN5 maintains an IPC close to 1, effectively hiding the coprocessor latency by executing independent instructions out of order.

In pipelined mode, LEN5 benefits from issuing multiple CLC instructions to the coprocessor, thanks to its 4-entry Reservation Station (RS). This allows overlapping the execution of multiple loop iterations, further enhancing performance.

Housekeeping Instructions

Next, housekeeping tasks were introduced by adding a function call within the loop (Listing 3.1c). This scenario simulates real-world applications where the processor performs additional tasks while waiting for the coprocessor.

Figure 3.5 illustrates the IPC results. The presence of housekeeping instructions improves the IPC for both cores, but LEN5 still outperforms CV32E40X significantly, achieving more than twice the IPC in some cases. The out-of-order execution allows LEN5 to utilize available instructions to mask the coprocessor latency effectively.

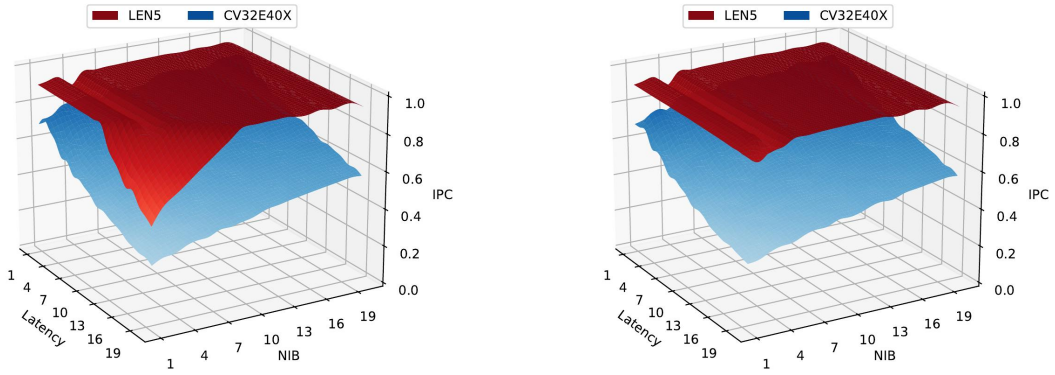


Figure 3.5: IPC comparison with housekeeping and without dependencies using iterative (left) and pipelined (right) CLC. Source: [23].

Dependent Loop Iterations

Scenarios with data dependencies between loop iterations (Listing 3.1b), such as accumulation or reduction operations, were also explored. In these cases, the CLC instruction in each iteration depends on the result from the previous iteration, limiting the ability to execute iterations in parallel.

Figures 3.6 and 3.7 present the IPC results for iterative and pipelined modes, respectively. The performance advantage of LEN5 diminishes compared to the independent case, but it still achieves higher IPC than the in-order core, particularly when housekeeping instructions are present.

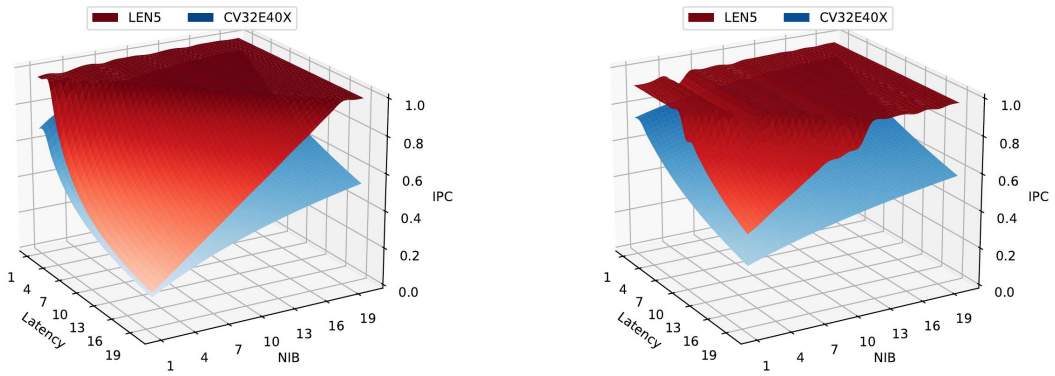


Figure 3.6: IPC comparison with dependencies using iterative CLC without housekeeping (left) and with housekeeping (right). Source: [23].

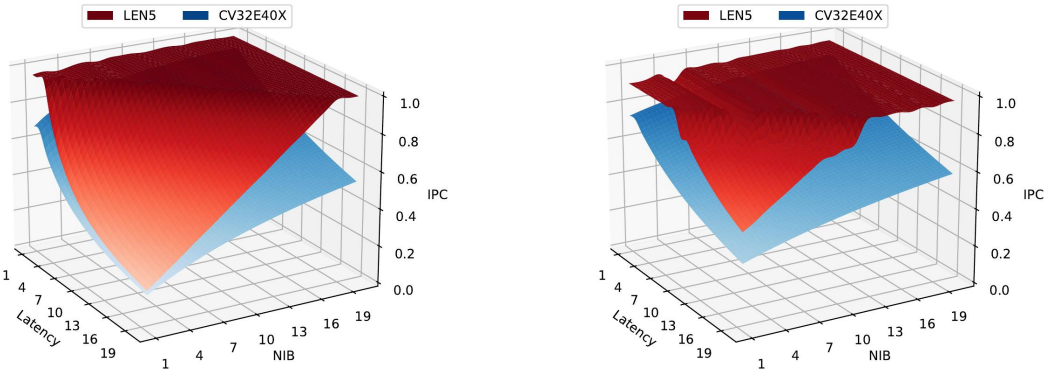


Figure 3.7: IPC comparison with dependencies using pipelined CLC without housekeeping (left) and with housekeeping (right). Source: [23].

Even with dependencies, LEN5 benefits from executing independent instructions out of order, partially masking the coprocessor latency. The in-order core, however, stalls while waiting for the coprocessor, resulting in lower IPC.

Effect of Long Coprocessor Latency

To assess the limitations of LEN5 when dealing with very long-latency instructions, the coprocessor latencies were set to exceed the size of the ROB in some of the experiments. The NIB was set to be equal to the coprocessor latency and varied the latency up to values much larger than the ROB size (32 entries).

Figure 3.8 shows the IPC degradation as the latency increases. When the coprocessor latency exceeds the ROB size, the issue stage stalls because the ROB cannot accept new instructions until older ones retire. This limitation causes the IPC to approach 0.5, similar to the in-order core.

The asymptotic IPC value can be approximated by:

$$\text{IPC} = \frac{1+N}{L+N-R+1} \tag{3.1}$$

where L is the coprocessor latency, N is the number of basic instructions, and R is the ROB size. As L and N approach infinity with $N = L$, the IPC approaches 0.5.

Addressing this limitation would require modifying the ROB to support out-of-order allocation and freeing of entries, allowing the issue stage to continue dispatching instructions

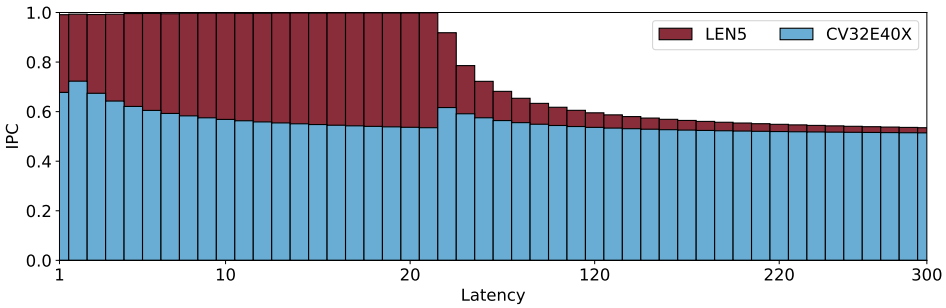


Figure 3.8: IPC comparison as the coprocessor latency and NIB increase, highlighting the impact of ROB size limitations. Source: [23].

even when older long-latency instructions are in flight. This enhancement is an area of ongoing development.

3.4.3 Real-World Benchmarking

To complement the synthetic benchmarks, LEN5 was evaluated using the Embench benchmark suite [65]. Figure 3.9 presents the IPC comparison between LEN5 and CV32E40P across various benchmarks.

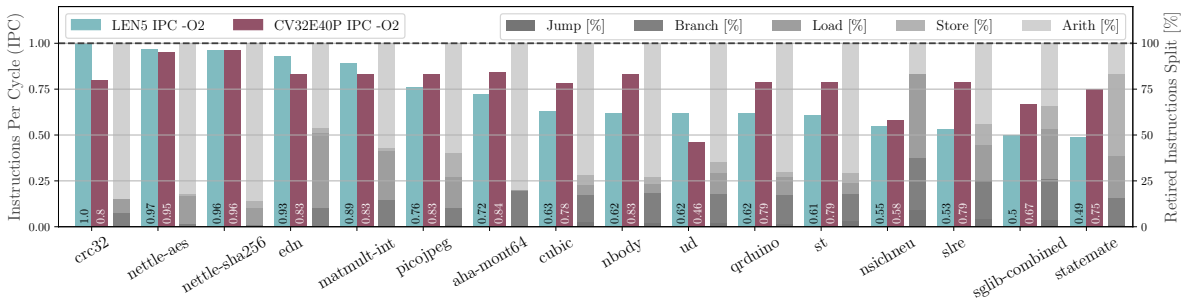


Figure 3.9: IPC comparison with CV32E40P over the Embench suite (colored bars) and executed instruction composition (grayscale). Source: [24].

LEN5 demonstrates higher IPC in benchmarks with low control-flow complexity, such as `crc32`, where it achieves an IPC of 1.0 and a performance improvement of over 20%. In compute-intensive tests like `edn` and `matmult-int`, LEN5 effectively hides memory access latencies, which constitute 30–40% of executed instructions.

However, in benchmarks with unpredictable and data-dependent branches, such as `aha-mont64`, LEN5 experiences lower IPC due to branch misprediction penalties. Despite this,

the 64-bit RISC-V ISA allows LEN5 to retire fewer instructions compared to the 32-bit CV32E40P, resulting in reduced execution time overall.

3.4.4 Discussion

The experimental results highlight the effectiveness of the proposed OoO microarchitecture in improving performance through latency hiding and dynamic scheduling. LEN5 consistently outperforms the in-order cores in scenarios with sufficient independent instructions to exploit ILP. The ability to integrate tightly coupled coprocessors without incurring significant stalls makes LEN5 suitable for modern embedded systems requiring both flexibility and efficiency.

The limitations observed with very long-latency instructions indicate areas for further optimization, such as enhancing the ROB management to support more aggressive out-of-order execution and commit strategies.

3.4.5 Conclusion

The performance assessment demonstrates that the LEN5 microarchitecture offers significant advantages over traditional in-order cores by leveraging OoO execution to mask instruction latencies and maximize resource utilization. Its modular and configurable design allows for adaptation to various application requirements, making it a promising solution for next-generation edge computing systems.

Chapter 4

Conclusion and Future Work

The growing demand for energy-efficient, high-performance computing at the edge has led to significant research and development in specialized hardware architectures. This thesis has explored two key directions: near-memory computing architectures, exemplified by NM-Carus, and a modular, Out-of-Order (OoO) CPU design, LEN5. Together, these architectures aim to overcome the limitations of traditional von Neumann systems, particularly in scenarios requiring intensive data processing, such as machine learning and artificial intelligence workloads.

In this chapter, we will summarize the major contributions of the thesis and reflect on the broader implications of the presented work for the future of edge computing.

4.1 Summary of Contributions

The key contributions of this thesis can be divided into two main areas: Near-Memory Computing (NMC) architectures and dynamically scheduled, OoO processors.

4.1.1 NM-Carus: A Programmable Near-Memory Computing Architecture

NM-Carus represents a novel approach to near-memory computing, specifically designed to bring the efficiency of Compute-In-Memory (CIM) architectures to edge devices. Its fully programmable, vector-capable design enables efficient execution of data-parallel operations, such as matrix multiplications and machine learning inference, directly within the memory subsystem. The major contributions of NM-Carus include:

- **Seamless integration into existing SoCs:** NM-Carus is designed as a drop-in replacement for standard SRAM banks, offering both memory and compute capabilities with minimal impact on system design and without requiring extensive modifications to the host architecture.
- **Custom RISC-V instruction set extension (`xvnmc`):** The newly proposed `xvnmc` vector extension allows NM-Carus to efficiently execute parallel operations directly within the memory, eliminating the need for explicit load/store instructions. This extension supports indirect register addressing, improving code compactness and flexibility for iterative operations.
- **Scalable vector processing unit (VPU):** NM-Carus includes a configurable vector processing unit that can scale with the number of SRAM banks, allowing system designers to trade off performance and area efficiency. This scalability is key to adapting NM-Carus to different edge applications with varying computational demands.
- **Energy-efficient data-parallel execution:** By performing computations directly on memory-stored data, NM-Carus reduces data movement, minimizing energy consumption while maintaining high computational throughput. This architecture is particularly suited for data-centric applications in resource-constrained environments.

4.1.2 LEN5: An Out-of-Order CPU Optimized for Coprocessor Utilization

The LEN5 processor introduces a modular, dynamically scheduled, OoO architecture optimized for edge applications. Its design allows for flexible integration with domain-specific coprocessors, enabling high performance across a variety of workloads while maintaining low power consumption. The contributions of LEN5 are as follows:

- **Dynamic scheduling and Out-of-Order execution:** LEN5 leverages OoO execution to maximize Instruction-Level Parallelism (ILP), allowing it to hide the latency of long-latency coprocessor instructions. This results in improved utilization of both the CPU and coprocessors, reducing idle time and increasing overall throughput.
- **Modular and configurable design:** The architecture of LEN5 is highly configurable, allowing it to be tailored for different application scenarios. Its modular design supports various Execution Units (EUs), providing flexibility in terms of performance, power, and area trade-offs.

- **Efficient handling of coprocessor offloading:** LEN5 is optimized for systems where coprocessors handle computationally intensive tasks via custom instruction set extensions. It ensures efficient execution by dynamically scheduling independent instructions in parallel with coprocessor tasks, reducing performance bottlenecks.
- **Performance improvement over in-order cores:** Experimental results show that LEN5 achieves significantly higher IPC compared to traditional in-order cores, particularly in workloads involving long-latency operations and complex control flows.

4.2 Limitations and Future Work

4.2.1 Limitations of Loosely Coupled Near-Memory Computing

In the approach to near-memory computing presented in Chapter 2, NM-Carus instances are integrated into a host system as loosely coupled CIM units together with traditional SRAM banks. In a typical deployment scenario, the host system is configured to acquire and store data directly inside the NM-Carus instance, which is accessible from the system bus like other traditional memory banks during the acquisition phase. In principle, this eliminates the need to move the data to process from an arbitrary region inside the system memory to the NM-Carus instance where the computation takes place. However, when the data size exceeds the capacity of a single NM-Carus instance, the data must be partially buffered into the main system memory. In this case, tiling techniques can be employed to partition the computation and transfer input and output data to and from the involved NM-Carus instances in chunks. While tiling is a viable solution, this approach partially defeats the purpose of near-memory computing, which aims to reduce data movements over the system bus. Having multiple NM-Carus instances can mitigate this issue to some extent, but it still requires explicit memory management and data partitioning in the application software. This makes the code inherently platform-dependent, therefore sacrificing portability and requiring additional effort during software development and application deployment.

This aspect is particularly critical because it introduces new data movements compared to a CPU-based system, where data is always stored and accessed from the main memory and directly transferred to the CPU for computation. Even in this case, the energy and throughput advantages of NM-Carus, which is several times more energy-efficient than a CPU-based system, are still partially preserved. However, this insight suggests that the performance benefits of a NM-Carus-based system are maximized when NM-Carus instances map all the on-chip

memory available to the system. In other words, one could use NM-Carus instances to implement the entire system's on-chip data cache, directly connected to the higher levels of the memory hierarchy. Since the cache virtually maps the entire system memory, NM-Carus instances can access any data acquired by the system and stored in its main memory. If the NM-Carus-based cache is not the Last-Level Cache (LLC) in a system's multi-level memory hierarchy, the data may still have to traverse higher levels of the memory subsystem to reach the NM-Carus instances. However, this would also be true when processing data inside the host CPU.

In such a memory architecture, the cache controller can be extended to load the required data when computation is offloaded. At the same time, the cache replacement policy can be tuned to maximize the amount of data reused inside an instance during long computation loops, thus reducing the amount of data that needs to be transferred back to the main memory. In other words, optimized tiling can be performed automatically by the cache controller without explicit management by the application software offloading the computation to the cache. Moreover, the normal cache data structures can be extended to keep track of what data (i.e., memory regions) is being updated or used as source operands by the offloaded computing instructions, functioning as a convenient abstraction level that implicitly synchronizes scalar memory accesses issued by the system CPU with the data processing performed by the NM-Carus instances. This approach eliminates the need for driver-based synchronization of the application software running on the host CPU with the NM-Carus instances. As a consequence, computation can be offloaded in the form of instructions triggering the execution of a specific compute kernel. The cache data structures ensure correct resolution of data hazards when bus transactions triggered by normal load and store instructions from the host CPU try to access data that is being processed by the NM-Carus instances. From the perspective of the host CPU, the cache doubles as a tightly coupled coprocessor whose execution engine is the array of NM-Carus instances also implementing the data cache memory. The application software or the host CPU is not required to know the details of the compute cache, as synchronization and data movement are implicitly handled by the cache controller and data structures, respectively.

These considerations represent the foundation of a tightly coupled in-cache architecture that wraps the near-memory computing approach introduced with NM-Carus in a convenient, instruction-based abstraction level. Such a CIM model hides the complexity of memory management from the application software and handles it by exploiting memory management units that are already present in traditional cache controllers, extended to enforce synchronization between bus transactions and offloaded CIM operations modifying the content of the memory. Such a system represents the ultimate outcome of the research conducted during the author's

Ph.D. studies and is being developed under the name of ARCANE in collaboration between the VLSI Laboratory at Politecnico di Torino and the Embedded Systems Laboratory at EPFL. The following section describes the ARCANE architecture in more detail, though early implementation results are not made available in this thesis as they are the subject of ongoing publications.

4.2.2 Tightly Coupled In-Cache Computing

A simplified top-level architecture of the ARCANE system is depicted in Fig. 4.1.

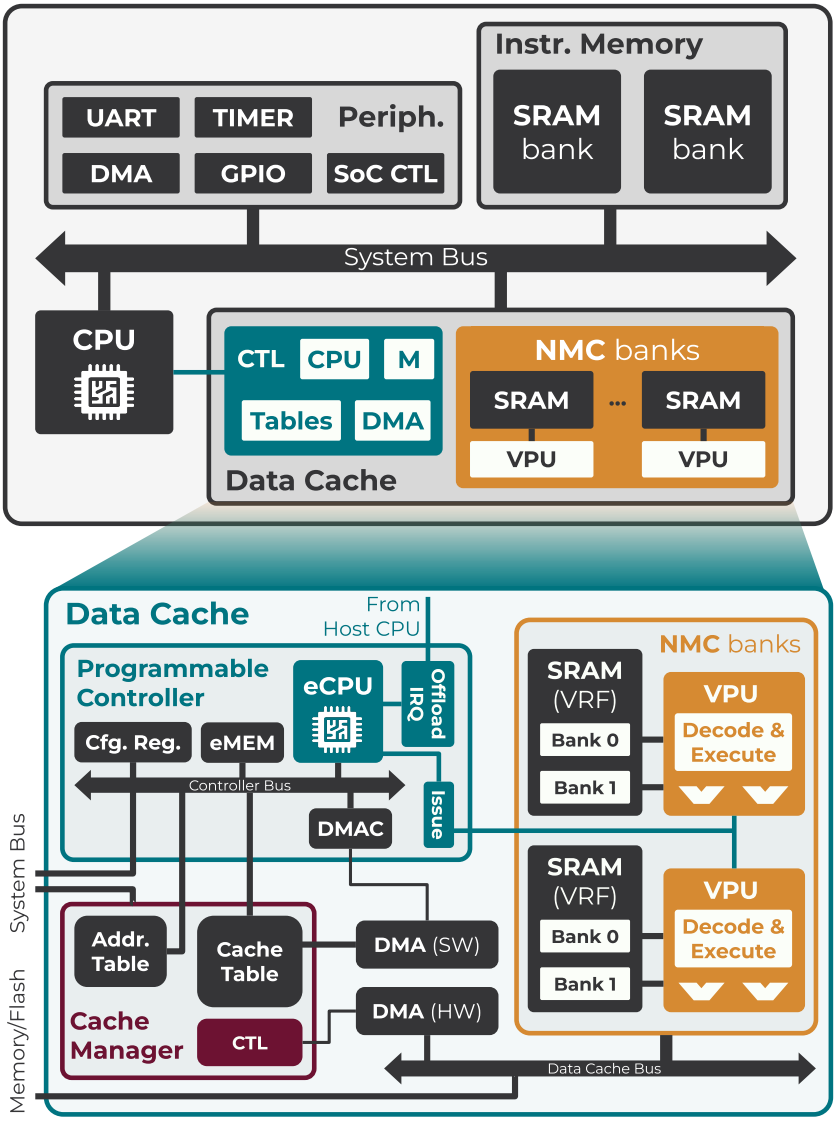


Figure 4.1: Top-level block diagram of the ARCANE system with detailed architecture of its NM-Carus-based data cache.

In ARCANE, the data cache is augmented with NM-Carus-based compute elements, transforming the cache into a programmable near-memory compute engine that accepts custom instructions from the host CPU, behaving as a traditional coprocessor from an application software point of view. At the core of ARCANE is a programmable cache controller based on a RISC-V CPU. Besides orchestrating the cache's memory management and data movement, the controller is responsible for offloading vector instructions to the NM-Carus instances implementing the cache's memory. Consequently, the CPU-based controller inside the original NM-Carus architecture presented in Section 2.2 was removed from the NM-Carus instances in ARCANE, which only integrate the VRF and VPU, resulting in a significant area reduction.

To maximize flexibility, the ARCANE controller handles instruction offloading in software. When a new instruction is received from the host CPU, a dedicated offloading interface samples the instruction opcode and operands into software-accessible status registers and triggers an interrupt to the cache controller. The associated Interrupt Service Routine (ISR) decodes the instruction and queues the execution of the associated compute kernel. A lightweight, simple Operating System (OS) implemented in the cache controller's firmware performs the necessary checks for data dependencies and hazards, fetches the required data if not yet available in the NM-Carus instances, and updates the cache data structures to ensure correct synchronization with normal bus transactions before initiating the execution of the queued vector kernel.

Since the NM-Carus architecture at the core of ARCANE has demonstrated great performance and energy efficiency when executing linear algebra operations, and such operations are very common in modern machine learning and signal processing workloads, the ISA of ARCANE was designed to use matrices as operands for the customizable compute instructions. The software-defined ISA offers unmatched flexibility and reconfigurability, and instructions can map arbitrarily complex operationssuch as entire convolutional layerswhich makes ARCANE an extreme example of a Complex Instruction Set Computer (CISC) architecture.

To enforce synchronization with the rest of the code, the ISA includes special instructions to communicate to the cache the intention to use certain memory regions as input and output operands for subsequent instructions. These instructions are defined as `xmr` for *matrix reserve*. The purpose of these instructions is to bind an arbitrary region inside the system address space (i.e., a memory array) to one of eight logical matrices. The logical matrices are then used as source and destination operands for subsequent computing instructions.

This approach differs from more traditional architectures where operands (scalar, vector, or matrix) are physically loaded into registers or scratchpad memories by explicit load and

store instructions. Instead, the `xmr` instructions preserve the CIM nature of the proposed cache architecture. Registering a new matrix does not imply any physical data movement by itself. The new matrix is simply registered by the cache controller's runtime in a list of known matrix bindings, together with its dimensions specified by the `xmr` operands.

Because a logical matrix is a simple alias for a certain address space, a registered matrix may be updated by either an offloaded computing instruction or a normal scalar memory operation issued by the host CPU. Therefore, hazards between these two types of instructions must be correctly handled by the cache. When a computing instruction is received and decoded, its source and destination operands are searched inside the list of registered matrices, and their status is updated to reflect that they are being used by an in-flight compute instruction. The address ranges corresponding to source and destination matrices are also inserted by the controller runtime into a hardware Address Table (AT). This table complements the normal cache table with the purpose of handling hazards when a bus transaction attempts to access a memory location bound to a logical matrix being used by an in-flight compute instruction. The address of each incoming bus transaction is checked against every entry inside the AT. If the address of a read transaction falls inside the address range of a destination matrix being written by an in-flight compute instruction, the transaction is delayed until the compute instruction completes to handle Read-After-Write (RAW) hazards. Read transactions on source matrices are normally executed as they do not represent any hazard. Finally, if the address of a write transaction falls inside the address range of a source or destination matrix being used by an in-flight compute instruction, it is delayed to prevent Write-After-Read (WAR) and Write-After-Write (WAW) hazards, respectively.

All checks against the AT are performed in parallel to serve incoming requests in a single cycle, as in a normal cache. Because each entry in the AT requires two hardware adders to perform the address check, the number of logical matrices that can be concurrently defined was limited to eight (`m0`–`m7`).

After updating the cache data structures with the information about the source and destination matrices of an offloaded instruction, the runtime can initiate the execution of the actual compute kernel inside the target NM-Carus instance. To do so, it reserves the necessary vector registers inside one of the NM-Carus instances, forcing the write-back of any dirty data previously mapped by those vector registers. The cache table is updated accordingly so that the reserved registers cannot be used to map new data.

Source matrices not available in the target instance are fetched from other instances or from higher levels in the memory hierarchy, depending on their availability, which the run-

time determines by consulting the cache table. The vector kernel implementing the offloaded instruction is then executed.

If supported by the kernel, vector instructions can be broadcast to multiple NM-Carus instances, each executing the same instruction sequence on different data chunks. This feature is particularly useful when the input data is larger than the available space in a single NM-Carus instance and allows processing multiple tiles in parallel. An example of this scenario is the computation of several output channels of a convolutional layer in a neural network.

Once the final result has been computed, the reserved vector registers inside the involved NM-Carus instances are released, and the status of the matrices inside the AT is updated, so any pending accesses can be resolved normally.

As a consequence of implicit synchronization, the application software running on the host CPU is relieved from manually managing the memory and its computing capabilities. This level of abstraction makes it possible for a programmer or a compiler to implement complex operations as a single instruction, without worrying about the details of the underlying hardware implementation. Of course, some of the complexity is moved to the firmware of the cache controller, but this firmware is expected to be modified only to update the supported ISA. Similar to the microarchitecture of a traditional CPU, the code implementing the ISA instructions accepted by the cache does not change as long as the ISA remains the same, with the added benefit that instructions can be added or, most importantly, fixed at any time through a firmware update.

The overhead of managing the cache data structures, decoding and accepting incoming instructions, and programming the integrated DMA controller to transfer the data required for the computation does have a performance cost. However, early experiments on a prototype RTL description of ARCANE suggest that this overhead can be limited to 10%–50% of the overall execution time of the offloaded instruction, depending on the type of operations, size of the input and output data, and the level of parallelism inside the NM-Carus instances (i.e., number of lanes in each NM-Carus instance).

Moreover, the same application binary can be deployed without any modification on any system implementing the included instructions in its cache controller. Also, since the design of the NM-Carus ISA follows the same implementation-agnostic approach as the RISC-V vector extension, the cache firmware can be written to be binary-compatible with ARCANE implementations with different cache sizes (i.e., number of NM-Carus instances) and levels of parallelism, ensuring convenient software portability and hardware scalability. This aspect partially addresses the issue of software compatibility and compilation or mapping effort.

4.2.3 New ILP Opportunities

As previously noted, the proposed cache design reveals how the software-defined instructions offloaded to the data cache can implement arbitrarily complex tasks, such as entire layers of a neural network. Consequently, the latency of such instructions can be substantial. This scenario offers new opportunities for exploiting ILP in the host core. Even when dealing with data-driven workloads, the application code usually consists of computationally intensive instructions interleaved with simple scalar instructions implementing the application control flow. Using an in-order core, the scalar instructions would be executed sequentially, waiting for the completion of long-latency offloaded instructions, as explained in Section 3.4.

In contrast, an out-of-order core can issue and execute independent scalar instructions in parallel with the long-latency cache operations, effectively hiding the latency of the latter. For example, the host CPU may compute the activation function of the output of a convolutional layer implemented by an instruction offloaded to the cache while the cache computes the output of a new layer of the network.

In this context, the LEN5 CPU is particularly well-suited as the host core. The cache handles dependencies among memory and offloaded instructions, while the LEN5 execution pipeline handles dependencies among scalar instructions and selects those that can be executed out of program order. Since loops are usually highly predictable in most computationally intensive workloads and exceptions are rare, the LEN5 pipeline can operate at full capacity most of the time.

4.2.4 Future Integration and Development

Although the integration of LEN5 into ARCANE was not completed during the author's Ph.D. studies, it remains an important direction for future research. The integration aims to fully leverage the ILP opportunities presented by the tightly coupled in-cache computing model of ARCANE, using LEN5's out-of-order execution capabilities to maximize performance.

The ongoing development of ARCANE, including the integration of LEN5, will focus on refining the cache controller's firmware to unlock higher levels of parallelism and minimize data movement among NM-Carus instances and through the memory hierarchy. Testing the resulting system after full physical implementation and while executing real-world applications is crucial to validate the performance and energy efficiency improvements brought by the proposed architecture, and to compare it against state-of-the-art solutions.

References

[1] P. E. Ceruzzi, *A history of modern computing*. MIT press, 2003.

[2] M. V. Wilkes, "The design of a practical high-speed computing machine. the edsac," *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, vol. 195, no. 1042, pp. 274–279, 1948.

[3] R. S. Barton, "A new approach to the functional design of a digital computer," in *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, 1961, pp. 393–396.

[4] F. Faggin, M. E. Hoff, S. Mazor, and M. Shima, "The history of the 4004," *Ieee Micro*, vol. 16, no. 6, pp. 10–20, 1996.

[5] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl, "Globally distributed content delivery," *IEEE Internet Computing*, vol. 6, no. 5, pp. 50–58, 2002.

[6] M. Armbrust, A. Fox, R. Griffith, *et al.*, "Above the clouds: A berkeley view of cloud computing," *Science*, vol. 53, pp. 07–013, 2009.

[7] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[8] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[9] M. Shafique, T. Theocharides, V. J. Reddy, and B. Murmann, "TinyML: Current progress, research challenges, and future roadmap," in *58th ACM/IEEE Design Automation Conf. (DAC)*, 2021, pp. 1303–1306.

[10] D. Sussillo, P. Kaifosh, and T. Reardon, "A generic noninvasive neuromotor interface for human-computer interaction," *bioRxiv*, 2024.

[11] A. Raychowdhury, C. Tokunaga, W. Beltman, M. Deisher, J. W. Tschanz, and V. De, "A 2.3 nJ/frame voice activity detector-based audio front-end for context-aware system-on-chip applications in 32-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 8, pp. 1963–1969, 2013.

[12] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, en. Morgan Kaufmann, Nov. 2017, Google-Books-ID: cM8mDwAAQBAJ.

[14] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st Conference on Computing Frontiers*, Association for Computing Machinery, 2004, p. 162.

[15] J. Backus, "Can programming be liberated from the von Neumann style? a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[16] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb. 2014, pp. 10–14.

[17] S. Srinivasa, A. K. Ramanathan, J. Sundaram, *et al.*, "Trends and opportunities for SRAM based in-memory and near-memory computation," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, 2021, pp. 547–552.

[18] G. Singh, L. Chelini, S. Corda, *et al.*, "Near-memory computing: Past, present, and future," *Microprocessors and Microsystems*, vol. 71, p. 102 868, 2019.

[19] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Enabling practical processing in and near memory for data-intensive computing," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19, 2019.

[20] C.-J. Jhang, C.-X. Xue, J.-M. Hung, F.-C. Chang, and M.-F. Chang, "Challenges and trends of sram-based computing-in-memory for ai edge devices," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 5, pp. 1773–1786, 2021.

[21] J. Heron and T. Chiang, "Magnetoelectrics and multiferroics: Materials and opportunities for energy-efficient spin-based memory and logic," *MRS Bulletin*, vol. 46, pp. 938–945, 2021.

[22] M. Caon, C. Choné, P. D. Schiavone, *et al.*, "Scalable and risc-v programmable near-memory computing architectures for edge nodes," 2024. arXiv: 2406.14263 [cs.AR].

[23] M. Caon, G. Masera, and M. Martina, "Ditching the queue: Optimizing coprocessor utilization with out-of-order cpus on compact systems on chip," *Electronics*, vol. 13, no. 15, 2024.

[24] M. Caon, V. Petrolo, M. Mirigaldi, F. Guella, G. Masera, and M. Martina, "Seeing beyond the order: A LEN5 to sharpen edge microprocessors with dynamic scheduling," in *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions*, 2024, pp. 47–50.

[25] F. Guella, E. Valpreda, M. Caon, G. Masera, and M. Martina, "Marlin: A co-design methodology for approximate reconfigurable inference of neural networks at the edge," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 71, no. 5, pp. 2105–2118, 2024.

[26] F. Guella, E. Valpreda, M. Caon, G. Masera, and M. Martina, "Temet: Truncated reconfigurable multiplier withăerror tuning," in *Applications in Electronics Pervading Industry, Environment and Society*, Cham, 2024, pp. 370–377.

[27] R.-V. Corp., *Risc-v 'v' vector extension, version 1.0*, 2022. [Online]. Available: https://github.com/riscv/riscv-v-spec.

[28] A. A. Khan, J. P. C. D. Lima, H. Farzaneh, and J. Castrillon, "The landscape of compute-near-memory and compute-in-memory: A research and commercial overview," 2024. arXiv: 2401.14428 [cs.AR].

[29] PULP Platform, *Pulp platform*, online, Accessed: Nov 8, 2024, 2019. [Online]. Available: https://pulp-platform.org.

[30] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, 1965.

[31] M. Gautschi, P. D. Schiavone, A. Traber, *et al.*, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[32] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conf. on computer vision and pattern recognition*, 2016, pp. 770–778.

[34] M. Kooli, A. Heraud, H.-P. Charles, *et al.*, "Towards a truly integrated vector processing unit for memory-bound applications based on a cost-competitive computational sram design solution," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 18, no. 2, pp. 1–26, 2022.

[35] S. Khoram, Y. Zha, J. Zhang, and J. Li, "Challenges and opportunities: From near-memory computing to in-memory computing," in *Proceedings of the 2017 ACM on International Symposium on Physical Design*, 2017, pp. 43–46.

[36] P. Deaville, B. Zhang, L.-Y. Chen, and N. Verma, "A maximally row-parallel mram in-memory-computing macro addressing readout circuit sensitivity and area," in *ESSCIRC 2021-IEEE 47th European Solid State Circuits Conference (ESSCIRC)*, 2021, pp. 75–78.

[37] L. He, X. Li, C. Xie, and Z. Song, "In-memory computing based on phase change memory for high energy efficiency," *Science China Information Sciences*, vol. 66, no. 10, p. 200 402, 2023.

[38] C.-X. Xue, W.-H. Chen, J.-S. Liu, *et al.*, "Embedded 1-mb reram-based computing-in-memory macro with multibit input and weight for cnn-based ai edge processors," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 1, pp. 203–215, 2019.

[39] W. Banerjee, "Challenges and applications of emerging nonvolatile memory devices," *Electronics*, 2020.

[40] K. Li, W. Yin, and Q. Liu, "A portable dsp coprocessor design using risc-v packed-simd instructions," in *IEEE ISCAS*, 2023, pp. 1–5.

[41] M. Platzer and P. Puschner, "Vicuna: A timing-predictable risc-v vector coprocessor for scalable parallel computation," in *33rd euromicro conference on real-time systems (ECRTS 2021)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[42] S. Soro, "Tinyml for ubiquitous edge ai," 2021. arXiv: 2102.01255 [cs.LG].

[43] F. Conti, D. Rossi, G. Paulin, *et al.*, "A 12.4tops/w @ 136gops ai-iot system-on-chip with 16 risc-v, 2-to-8b precision-scalable dnn acceleration and 30%-boost adaptive body biasing," in *IEEE ISSCC*, 2023, pp. 21–23.

[44] M. Sinigaglia, L. Bertaccini, L. Valente, *et al.*, "Echoes: A 200 gops/w frequency domain soc with fft processor and i2s dsp for flexible data acquisition from microphone arrays," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023, pp. 1–5.

[45] R. Gauchi, V. Egloff, M. Kooli, *et al.*, "Reconfigurable tiles of computing-in-memory sram architecture for scalable vectorization," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '20, 2020, pp. 121–126.

[46] OpenHW Group, *Openhw group cv32e40x user manual*, online, Accessed: Nov 8, 2024, 2023. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/index.html.

[47] OpenHW Group, *Openhw group specification: Core-v extension interface (cv-x-if)*, online, Accessed: Nov 8, 2024, 2023. [Online]. Available: https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/latest/index.html.

[48] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2020.

[49] S. Machetti, P. D. Schiavone, T. C. Müller, M. Peón-Quirós, and D. Atienza, "X-heep: An open-source, configurable and extendible risc-v microcontroller for the exploration of ultra-low-power edge accelerators," 2024. arXiv: 2401.05548 [cs.AR].

[50] W. A. Simon, Y. M. Qureshi, M. Rios, A. Levisse, M. Zapater, and D. Atienza, "Blade: An in-cache computing architecture for edge devices," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1349–1363, 2020.

[51] Y. Wang, M. Yang, C.-P. Lo, and J. P. Kulkarni, "30.6 Vecim: A 289.13GOPS/W RISC-V vector co-processor with compute-in-memory vector register file for efficient high-performance computing," in *2024 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 67, 2024, pp. 492–494.

[52] J.-P. Noel, M. Pezzin, R. Gauchi, *et al.*, "A 35.6 tops/w/mm^2 3-stage pipelined computational sram with adjustable form factor for highly data-centric applications," *IEEE Solid-State Circuits Letters*, vol. 3, pp. 286–289, 2020.

[53] H. Daoud and M. A. Bayoumi, "Efficient epileptic seizure prediction based on deep learning," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 5, pp. 804–813, 2019.

[54] S. Hoshino and Y. Kubota, "Mobile robot motion planning through obstacle state classifier," in *2023 62nd Annual Conference of the Society of Instrument and Control Engineers (SICE)*, 2023, pp. 120–126.

[55] Y. Wang, J. Jiang, S. Li, *et al.*, "Decision-making driven by driver intelligence and environment reasoning for high-level autonomous vehicles: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 10, pp. 10 362–10 381, 2023.

[56] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini, "XpulpNN: Accelerating quantized neural networks on RISC-V processors through ISA extensions," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 186–191.

[57] F. Conti, G. Paulin, A. Garofalo, *et al.*, "Marsellus: A heterogeneous risc-v ai-iot end-node soc with 2–8 b dnn acceleration and 30%-boost adaptive body biasing," *IEEE Journal of Solid–State Circuits*, vol. 59, no. 1, pp. 128–142, 2024.

[58] H. Genc, S. Kim, A. Amid, *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 769–774.

[59] R.-V. International, *Risc-v cryptography extensions, volume ii: Vector instructions*, online, Accessed: Nov 8, 2024, 2023. [Online]. Available: https://github.com/riscv/riscv-crypto.

[60] A. Dolmeta, M. Mirigaldi, M. Martina, and G. Masera, "Implementation and integration of keccak accelerator on risc-v for crystals-kyber," in *Proc. of the 20th ACM Int. Conf. on Computing Frontiers*, 2023, pp. 381–382.

[61] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[62] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[63] R. Alves, A. Ros, D. Black-Schaffer, and S. Kaxiras, "Filter caching for free: The untapped potential of the store-buffer," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 436–448.

[64] P. D. Schiavone, F. Conti, D. Rossi, *et al.*, "Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.

[65] Embench, *Embench: Open benchmarks for embedded platforms*, online, Accessed: Nov 8, 2024, 2019. [Online]. Available: https://github.com/embench/embench-iot.

Appendix A

Code Snippets

A.1 Matrix Multiplication Kernel

The code in Listing 3 snippet shows a simple matrix multiplication algorithm implemented in RISC-V assembly language for a simple, scalar CPU.

RV32I Matrix Multiplication (any size, 32-bit)

```
_start:
  # ... initialization code
  li    t0, 0           # Initialize row index for A
RowLoop:
  beq   t0, a0, End     # Check if all rows processed
  li    t1, 0           # Initialize column index for B
ColLoop:
  beq   t1, a1, NextRow  # Check if all columns processed
  li    t2, 0           # Initialize accumulator
  li    t3, 0           # Initialize element index
ElemLoop:
  beq   t3, a2, Accumulate # Check if all elements processed
  # Load elements from A and B
  lw    t4, 0(addr_A)
  lw    t5, 0(addr_B)
  mul   t6, t4, t5      # Multiply elements
  add   t2, t2, t6      # Accumulate result
  addi  addr_A, addr_A, 4
  addi  addr_B, addr_B, 4
  addi  t3, t3, 1
  j     ElemLoop
Accumulate:
  sw    t2, 0(addr_R)  # Store result
  addi  addr_R, addr_R, 4
  addi  t1, t1, 1
  j     ColLoop
NextRow:
  addi  t0, t0, 1
  j     RowLoop
End:
```

Listing 3: Assembly code for matrix multiplication on a standard RISC CPU.