POLITECNICO DI TORINO
Repository ISTITUZIONALE

Scalable and RISC-V Programmable Near-Memory Computing Architectures for Edge Nodes

Original
Scalable and RISC-V Programmable Near-Memory Computing Architectures for Edge Nodes / Caon, M.; Chone, C.; Schiavone, P. D.; Levisse, A.; Masera, G.; Martina, M.; Atienza, D.. - In: IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING. - ISSN 2168-6750. - ELETTRONICO. - 13:3(2025), pp. 1003-1018. [10.1109/TETC.2025.3555869]

Availability:
This version is available at: 11583/3011175 since: 2026-05-21T11:52:57Z

Publisher:
IEEE

Published
DOI:10.1109/TETC.2025.3555869

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

01 June 2026

Scalable and RISC-V Programmable Near-Memory Computing Architectures for Edge Nodes

Michele Caon, Clément Choné, Pasquale Davide Schiavone, Alexandre Levisse, Guido Masera *Senior member, IEEE*, Maurizio Martina *Senior member, IEEE*, David Atienza *Fellow, IEEE*

***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 necessitates stringent constraints on energy efficiency, which traditional von Neumann architectures struggle to meet. The Compute-In-Memory (CIM) paradigm has emerged as a better candidate due to its efficient exploitation of the available memory bandwidth. However, existing CIM solutions require a high implementation effort and lack flexibility from a software integration standpoint. This work proposes a novel, software-friendly, general-purpose, and low-integration-effort Near-Memory Computing (NMC) approach, paving the way for the adoption of CIM-based systems in the next generation of edge computing nodes. Two architectural variants, NM-Caesar and NM-Carus, are proposed and characterized to target different trade-offs in area efficiency, performance, and flexibility, covering a wide range of embedded microcontrollers. Post-layout simulations show up to $28.0\times$ and $53.9\times$ lower execution time and $25.0\times$ and $35.6\times$ higher energy efficiency at system level, respectively, compared to the execution of the same tasks on a state-of-the-art RISC-V CPU (RV32IMC). NM-Carus achieves a peak energy efficiency of $306.7\,\mathrm{GOPS/W}$ in 8-bit matrix multiplications, surpassing recent state-of-the-art in- and near-memory circuits.**

***Index Terms*—Near-Memory Computing, Edge Computing, RISC-V, Embedded Systems, Microcontrollers.**

I. Introduction

THE last decades have witnessed a shift in the computing paradigm towards data-driven workloads, such as Artificial Neural Networks (ANNs). These computationally expensive algorithms offer advanced functionalities in a wide range of applications by exploiting an exponentially increasing volume of data from the internet and Internet of Things (IoT) devices. In this context, edge computing has gained traction, driving the exploration of innovative computing architectures that provide superior energy efficiency and performance compared to traditional systems. These advances are crucial in overcoming the throughput, latency, energy, power, and privacy limitations of centralized computing infrastructures while meeting the increased computational demands for real-time data processing in fields such as biosignals [1], audio [2], video, etc. The accessible manufacture and deployment of low-power, high-performance, and software-friendly edge computing devices for the new era of Artificial Intelligence (AI) is the main motivation behind the research presented in this paper. The traditional von Neumann architecture, which has been the backbone of computing systems for decades, is inherently inefficient for data-intensive workloads due to the constant need to move data and instructions between the system memory hierarchy and the Central Processing Unit (CPU) registers [3], [4]. This is exacerbated by Static Random-Access Memory (SRAM) integration technologies not keeping pace with logic scaling, an issue known as the *memory wall* [5]. As a consequence, SRAM accesses typically account for $100\times$ the energy of the arithmetic operations in the CPU, as noted by J. Hennessy and D. Patterson [6]. The *Compute-In-Memory (CIM)* paradigm has been proposed as a solution to this problem in the form of In-Memory Computing (IMC) and Near-Memory Computing (NMC) [7]–[9]. Their common intuition is to move processing closer to the data to alleviate the instruction fetch overhead and the pressure on the system bus significantly while optimizing the utilization of the available internal memory bandwidth.

M. Caon is with the Embedded Systems Laboratory (ESL), EPFL, 1015 Lausanne, Switzerland and with the Very Large Scale Integration Laboratory (VLSI Lab), DET, Politecnico di Torino, 10129 Torino, Italy (e-mail: michele.caon@epfl.ch). C. Choné, P. D. Schiavone, A. Levisse and D. Atienza are with the Embedded Systems Laboratory (ESL), EPFL (e-mail: clement.chone@epfl.ch; davide.schiavone@epfl.ch; alexandre.levisse@epfl.ch; david.atienza@epfl.ch). G. Masera and M. Martina are with the Very Large Scale Integration Laboratory (VLSI Lab), DET, Politecnico di Torino (e-mail: guido.masera@polito.it; maurizio.martina@polito.it).

M. Caon and C. Choné contributed equally to this work.

The preserved versatility of flexible memory access in near-memory systems suits the semantics of traditional programmable systems, where data processing is expressed through instructions operating on data from various memory locations. Consequently, the optimizations available in traditional von Neumann architectures, such as Single Instruction Multiple Data (SIMD) and vector operations, can be easily applied to CIM systems. This was demonstrated in [10], where the energy efficiency of a vector coprocessor based on Ara [11] was further enhanced by integrating parts of the arithmetic circuitry within the Vector Register File (VRF) memory banks.

Controlling operations on NMC Integrated Circuits (ICs) remains a challenge. The simplest control strategy involves streaming individual micro-operations to the NMC unit using an external CPU or Direct Memory Access (DMA) controller. This approach is very area-efficient, does not rely on a dedicated external controller, and scales well for regular packed-SIMD applications. However, complex runtime control requires either the CPU to spend significant time encoding such operations at runtime, decreasing system efficiency, or relying on predefined command sequences, resulting in substantial code size overhead. In both cases, additional instructions and data must be fetched from the main memory before sending commands to the NMC ICs, reducing energy efficiency. Alternatively, custom controllers can assemble the

necessary commands in hardware, increasing energy efficiency and throughput at the cost of reduced flexibility. To avoid microcontrolling the NMC unit, more complex and versatile controllers can be employed. For example, a dedicated CPU can be coupled with a NMC block to run small programs via a custom controller memory interface, avoiding standard load/store memory-mapped operations. This enables the execution of more complex programs independently from the host system, similar to a conventional accelerator, with the advantage of processing private data only with closer processing elements, resulting in fewer data movements. However, such controllers trade flexibility for increased area and power, making them unsuitable for ultra-low-power, edge-oriented Systems-on-Chip (SoCs).

In this work, we elaborate on this concept to present two architectural variations of a novel NMC approach and their corresponding IC implementations, which target different requirements at the system and application level, such as complexity, performance, and integration with existing platforms:

- *NM-Caesar*, an area-efficient, SIMD-enabled NMC unit micro-controlled by the host system, targeting regular TinyML[1] benchmarks (i.e., AI-based biomedical application kernels with a regular control flow) such as min/max search algorithms for peak detection [12], and lightweight ANNs [13] used in arrhythmia detection.
- *NM-Carus*, a fully-autonomous, vector-capable RISC-V-based programmable NMC unit targeting highly-parallel and complex TinyML applications [14] across domains involving performance-critical and computationally-intensive workloads (i.e., Deep Neural Networks (DNNs) [15], [16] or tasks with a data-dependent control flow).

Both NM-Carus and NM-Caesar are designed to be energy-efficient computing solutions as close as possible to a drop-in alternative to a traditional embedded SRAM bank both from a physical and functional standpoint. They offer an SRAM-compatible interface to the host system and have a functionally transparent *memory* operating mode alongside their *computing* mode. Their focus on programmability and ease of integration represents a significant advancement to address the challenges that prevent available CIM architectures from achieving broader adoption, as highlighted in [17].

In particular, the main contributions of the presented work on emerging computing paradigms are as follows:

- A software-friendly, low-cost, and low-integration-effort approach to NMC in the context of general-purpose, low-power edge devices. Its effectiveness is demonstrated through the implementation of two architectural variations targetting various classes of embedded SoCs.
- Conception of a code-efficient RISC-V custom Instruction Set Architecture (ISA) extension to support a flexible set of vector operations on programmable CIM architectures. The proposed extension provides a vector-based view of the host's compute memory without resulting in the overhead of explicit vector load/store operations.
- An in-depth, quantitative, and vertical analysis of the impact and benefits of replacing conventional SRAM banks with the proposed NMC macros in low-power Microcontroller Units (MCUs).

The remainder of this paper is organized as follows. Section II reviews existing works on improving the performance and energy efficiency of edge-computing systems, focusing on CIM devices. Sections III and IV detail the architecture and physical implementation of NM-Caesar and NM-Carus. Section V describes the experimental setup, benchmarking methodology, and results. Section VI summarizes the main conclusions for the NMC paradigm emerging from this work.

II. Related Works

To overcome the efficiency limitations of traditional von Neumann systems in handling data-centric workloads, recent research has explored alternative and more efficient computing paradigms. Among these, application- or domain-specific accelerators and CIM devices are the most prominent ones, and the closest to the proposed NMC architecture presented in this work.

A. Energy-efficient Edge Accelerators

Fixed-function Application-Specific Integrated Circuits (ASICs) have been proposed for maximum performance and energy efficiency, whose predominant examples are ANN [18]–[20] accelerators, specialized Digital Signal Processors (DSPs) for real-time data elaboration kernels such as Fast Fourier Transforms (FFTs) [21], or cryptographic coprocessors [22]. These ICs leverage highly parallel computing engines and optimized memory management units that maximize data reuse to achieve superior performance and energy efficiency, at the cost of reduced flexibility.

For scenarios requiring greater versatility, reconfigurable solutions such as embedded Field-Programmable Gate Arrays (FPGAs) [23] and Coarse-Grained Reconfigurable Architectures (CGRAs) [24], [25] can handle a wider range of workloads, although with a higher area and power overhead compared to ASICs. From a functional perspective, they often suffer from suboptimal resource utilization due to the complexity of the application mapping process, especially in on-line-reconfigurable CGRAs where spatial and temporal mapping constraints coexist. More conventional architectures preserve programmability while overcoming the instruction fetch overhead by offloading tasks to on-chip domain-specific processors [26], Vector Processing Unit (VPU) [11], [27], or multi-core clusters with optimized data interconnects [28]. These subsystems efficiently execute critical parts of the workload, reducing the pressure on the system bus and the memory hierarchy, thereby improving overall performance and energy efficiency. However, they still rely on a dedicated local memory that requires specific management in the application software, resulting in additional instructions and data movement. To address these limitations and leverage the benefits of domain-specific architectures while significantly reducing data movement, our work aims to maintain the flexibility and ease of implementation of programmable accelerators

[1]TinyML applications here refer to compact machine-learning algorithms deployed on energy- and area-constrained edge devices.

while exploiting the advantages of the aforementioned CIM paradigm. With this objective, it implements an innovative integration model where compute-capable drop-in replacements for conventional SRAMs can be arbitrarily programmed to process data in place, with minimal impact on the host's software stack and physical characteristics.

B. Compute Memories

IMC and NMC circuits enable concurrent data access from several memory banks and, in the case of IMC devices, also exploit the internal parallelism and structure of the memory arrays to enhance arithmetic throughput with lower area overhead compared to conventional memory-mapped accelerators. Over the years, this concept has been applied at various levels of the memory hierarchy. *Embedded Non-Volatile Memory (ENMV)* technologies such as Magnetoresistive Random Access Memory (MRAM) [29], Phase-Change Memory (PCM) [30], and Resistive Random Access Memory (RRAM) [31] have emerged as next-generation memory solutions enabling fast, power-efficient Multiply-and-Accumulate (MAC) operations in the analog domain, making them suitable for applications tolerating inexact results, such as ANNs. However, their stochastic nature also leads to reliability, uniformity, and portability issues, making them unsuitable for memory-critical applications [32]. Consequently, research has also focused on DRAM- and SRAM-based solutions [33], [34]. *Dynamic Random Access Memories (DRAMs)* offer high density and large storage capacity, reducing data transfers from other memory devices, which is beneficial for data-intensive CIM scenarios. However, their high memory access energy limits their use in energy-constrained TinyML systems.

Conversely, *Static Random-Access Memories (SRAMs)* provide lower-energy and faster memory accesses, motivating studies on SRAM-based IMC or NMC architectures [35]. As a result, SRAM has become the predominant technology for data-intensive applications [17]. In IMC devices, the integration of arithmetic circuitry directly within the memory array allows fine-grained optimization, producing superior performance and energy efficiency [36]. However, this approach is highly invasive at the system level, introducing significant portability challenges and imposing strict data placement constraints that degrade application-level performance, as demonstrated in Section V-C. In contrast, NMC architectures place digital processing units outside the memory periphery. This choice leverages a standard digital integration flow, reducing implementation effort and enhancing portability across technology nodes. These advantages motivated the adoption of an NMC approach for the architectures proposed in Section III. Hybrid IMC/NMC solutions [7], [10], [34], [37] have been explored as alternatives. In [10], Wang et al. propose Vecim, a RISC-V vector coprocessor leveraging a CIM-based VRF. Its modular design features configurable vector lanes that implement in-memory MAC operations through bit-line computing. This design achieves a state-of-the-art peak energy efficiency of $289.1\,\mathrm{GOPS/W}$ in $65\,\mathrm{nm}$ CMOS technology during an 8-bit matrix multiplication. In comparison, the proposed NM-Carus macro achieves $306.7\,\mathrm{GOPS/W}$ under similar conditions to a full NMC design giving priority to memory capacity and reduced data movement over high parallelization of processing elements, as explained in Section III-B. Kooli et al. [34] present a Computational SRAM (C-SRAM) solution combining IMC and NMC to perform vector operations on memory rows. However, this approach suffers from low bitcell density due to data replication and the use of larger two-port memories. Gauchi et al. [37] propose a workload-aware, reconfigurable multi-tile C-SRAM architecture to enhance vectorization, albeit with higher area and power overhead, reduced versatility, and suboptimal utilization of computing resources on small kernels. C-SRAM acts as a memory-mapped accelerator that is micro-controlled with commands encoded in bus write transactions. NM-Caesar uses a similar control strategy while achieving higher bitcell density and energy efficiency thanks to smaller single port memories at the cost of no support for vector operations and lower throughput, as reported in Section V-C.

From a technological standpoint, these hybrid approaches mitigate many challenges associated with fully IMC-based solutions. By placing most of the processing hardware outside the memory arrays, they maintain a clear separation between physical memory and arithmetic circuitry. This separation simplifies validation and verification while enabling faster architectural exploration. However, they still present integration challenges at the system level, which are addressed in this work. As discussed in [17], strict data placement constraints, and the need for dedicated control schemes significantly hinder the development of a robust software deployment ecosystem. These factors, rather than technological limitations alone, remain a major barrier to the widespread adoption of CIM devices in real-world applications. The CIM paradigm introduced in this paper aims to overcome these limitations by enabling transparent integration of compute-memory elements within the memory subsystem of the host SoC (Section III). The proposed NMC architectures provide a low-impact integration effort from both software and hardware perspectives (Section IV), combining the efficiency of SRAM-based CIM with the flexibility and portability of programmable accelerators.

III. Near-Memory Computing Architectures

The main goal of the proposed NMC paradigm is to bring the efficiency of CIM to existing SoCs with minimal integration effort, providing an easy-to-deploy solution to the growing computational needs of next-generation edge computing devices. To achieve this, the NMC device must adhere to two fundamental requirements: (1) functionally, it is part of the host system's memory space and should operate like a conventional memory during normal use; (2) from a physical implementation perspective, it represents a direct replacement for a standard SRAM memory bank, with similar area and timing characteristics. An overview of the resulting integration model is shown in Fig. 1.

For straightforward functional integration (1), the proposed NMC devices connect to the system bus with a slave interface, similar to conventional memories. A transparent *memory* mode provides read and write access to the memory content, while a *computing* or *configuration* mode enables streaming

or programming of arbitrary processing kernels. The data is processed in place within the IC private memory, which is managed and populated by the host CPU or DMA engine. The computation results are directly accessible by the host system, eliminating additional data movement. All interactions take place on a memory-like interface, whose data write bus is used to encode and stream instructions when in *computing* mode. An additional `imc` pin, routed to a dedicated configuration register in the host system, switches the operating mode and is controlled by software, without altering the bus protocol. Unlike other implementations where the address bus encodes instructions, this solution avoids fragmentation of the address space and consequent mapping constraints when linking the firmware. The cost of setting and resetting the `imc` configuration register is negligible in common edge-computing workloads, where a long burst of operations is executed after writing and before reading back the memory entries.

The physical integration (2) of the proposed NMC ICs is facilitated by the standard digital implementation flow for logic and the use of foundry-provided SRAM compilers for internal memories. This improves technological portability compared to IMC solutions that rely on custom memory arrays. The control and data processing units are carefully designed to maintain the timing characteristics of a reference conventional SRAM memory with the same capacity, while also minimizing the area overhead. Particular attention is paid to preserving the reference input and output delays, ensuring compatibility when replacing a standard memory bank with the NMC device.

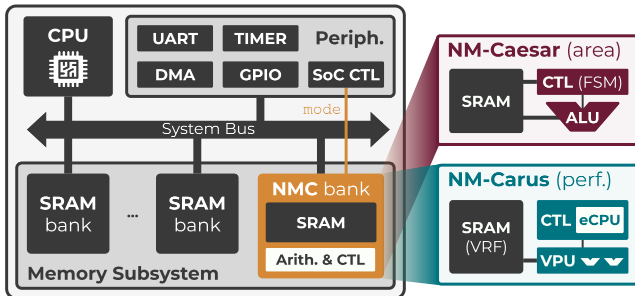


Figure 1: Top-level block diagram of a NMC-enhanced MCU hosting NM-Caesar (area-critical implementations) or NM-Carus (performance-oriented applications) as part of its memory subsystem.

Having established the system integration model and its implication in functional and physical aspects of the design, it is possible to motivate the design choices behind the proposed NMC architectures. The chosen fully digital NMC paradigm enables the development of configurable and scalable architectures, possibly covering diverse performance requirements. However, the physical constraints of the smallest class of IoT devices conflict with the level of flexibility that brings the most advantages to larger, performance-oriented systems. To address this trade-off, two distinct NMC architectures were designed and implemented. NM-Caesar targets low cost SoCs with strict area and power constraints, such as near-sensor MCUs. NM-Carus, on the other hand, fits well with those SoCs where a larger area budget can be allocated for increased performance and extended functionalities. The different design goals are mainly reflected in the control strategy adopted in each device. NM-Caesar relies on the host system to stream the necessary SIMD instructions from a reduced ISA. In contrast, NM-Carus integrates a RISC-V-based controller supporting a custom CIM-oriented ISA, paired with a novel and scalable VPU for maximum flexibility and tunable parallelism at the data level. Since both architectures are designed for streamlined software deployment, their ISA and microarchitecture were tailored to support standard data types (8-, 16-, and 32-bit integers). Given the limited area budget, support for application-specific lower-precision data types was considered but not implemented, as many edge applications, such as health monitoring and audio processing, still rely on standard data types.

The following sections describe the proposed architectures, motivating their ISAs and microarchitecture.

A. NM-Caesar

Minimizing the area overhead and power consumption of the NMC controller and arithmetic circuitry is the top-priority design criteria for the definition of NM-Caesar's ISA and microarchitecture, discussed in Sections III-A1 and III-A2.

1) Instruction Set Architecture: Designed for low-power edge computing applications, NM-Caesar ISA comprises operations that are common in Machine Learning (ML) algorithms such as Convolutional Neural Networks (CNNs) and support vector machines: arithmetic and logic operations that include multiplication, MAC, dot product, minimum and maximum selection, and shift operations to support fixed-point arithmetic. A summary of the supported instructions is shown in Table I.

Table I: NM-Caesar instruction listing and accumulator operations. All the instructions are element-wise unless otherwise specified.

Instruction				Acc.	Description
Arithmetic-Logic Instructions					
`{AND,OR,XOR}`	`dest`	`src1`	`src2`	-	Bitwise logic
`{ADD,SUB,MUL}`	`dest`	`src1`	`src2`	-	Addition, subtraction, mult.
`MAC_INIT`		`src1`	`src2`	clear	Multiply-add initialization
`MAC`		`src1`	`src2`	update	Multiply-add
`MAC_STORE`	`dest`	`src1`	`src2`	update	Multiply-add writeback
`DOT_INIT`		`src1`	`src2`	clear	Word-wise dot-product init.
`DOT`		`src1`	`src2`	update	Word-wise dot-product
`DOT_STORE`	`dest`	`src1`	`src2`	update	Word-wise dot-product wb.
`{SLL,SLR}`	`dest`	`src1`	`src2`	-	Logic shift left and right
`{MIN,MAX}`	`dest`	`src1`	`src2`	-	Min./max. selection
Configuration Instructions					
`CSRW`	`bitwidth`			-	Set operand bitwidth in CSR

When in *computing* mode, NM-Caesar interprets the write transactions on the bus as instructions. The opcode is encoded in the six most significant bits of the data bus, followed by the word address of the two source operands, relative to NM-Caesar base address and 13-bit wide on 32-bit systems, for a total of 32 KiB of addressable space. The address bus encodes the target address (destination address) as in normal accesses. For example, suppose that NM-Caesar is mapped at base address `BASE`, the command to sum together the words at offset `SRC1` and `SRC2` and store the result at offset `DEST` can be assembled and issued online with the following C code:

```
*(BASE + DEST << 2) = ADD << 26 | SRC2 << 13 | SRC1;
```

Alternatively, an in-house domain-specific compiler can be used to assemble predefined sequences of NM-Caesar instructions that implement specific kernels. These are compiled and

embedded into the host system and sent to NM-Caesar by the host CPU or DMA controller during execution. To efficiently deal with different data types, all NM-Caesar instructions are packed-SIMD. The data type is statically configured in a Control and Status Register (CSR) updated by a dedicated instruction to avoid repeated instruction encodings.

2) Microarchitecture: As shown in Fig. 2, NM-Caesar is composed of two single-port SRAM macros, an integer Arithmetic Logic Unit (ALU), and a controller that interfaces with the system bus, decodes incoming instructions, and schedules internal memory accesses and arithmetic operations. Internally, NM-Caesar can access both memory banks simultaneously to increase the bandwidth towards the ALU. Splitting the data memory into two smaller banks has a minor overhead in area and power due to the partially replicated periphery, yet it is more area-efficient than using a single dual-port memory for the same throughput.

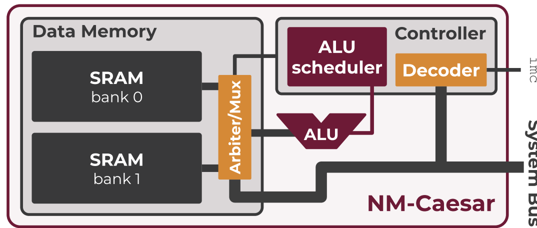


Figure 2: Top-level block diagram of NM-Caesar.

The received instructions go through four phases, handled by a 2-stage pipeline as shown in Fig. 3: (1) the instruction is decoded (controller *dec* stage), and the source and destination addresses are buffered; (2) in the next cycle, the source operands are fetched from the two memory banks (controller *fetch* stage); (3) the operands are received in the following cycle when the necessary ALU operation is scheduled. A new external request can be decoded at this point; (4) once the ALU provides the result, it is written back to the destination memory location. To minimize the logic area and meet the same timing constraints of the reference 32 KiB SRAM, a multi-cycle, 32-bit, SIMD integer ALU was selected. Choosing a single-cycle architecture would not improve performance as read and write contentions would occur at the memory ports. The ALU design is based on that of the CV32E40P core [38] ALU. NM-Caesar inherits its partitioned multi-precision adder supporting 8-, 16-, and 32-bit operations, here implemented with relaxed timing constraints to meet a 2-cycle propagation delay. The adder is used by the addition, subtraction, and minimum or maximum selection instructions. Unlike CV32E40P, which employs several parallel multipliers to perform single-cycle MAC and dot products, NM-Caesar employs four 17-bit multipliers and four adders producing one 8-bit, two 16-bit, and four 8-bit results every two cycles. Although no data placement constraints exist in NM-Caesar, the throughput is reduced to one operation every three cycles when both source operands come from the same memory bank, where they are accessed sequentially.

B. NM-Carus

NM-Carus architecture is designed to leverage the increased data-access efficiency brought by the NMC paradigm while

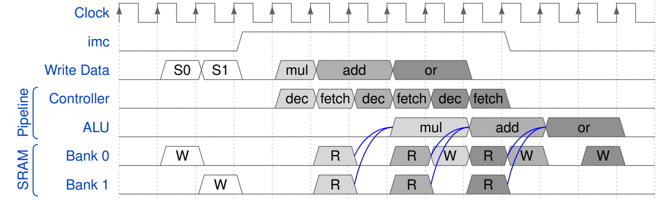


Figure 3: Example timing diagram of NM-Caesar running two normal write operations ($S0$ and $S1$) and three instructions.

also: (1) offering the advantages in terms of flexibility, scalability, and instruction fetch overhead that are typically offered by programmable vector machines and (2) relying on a scalable microarchitecture with configurable data-level parallelism. To achieve (1) and (2), NM-Carus integrates an innovative controller based on a small RISC-V CPU, placed near the internal SRAM banks and called embedded CPU (eCPU) to distinguish it from the external host system's processor. A small memory, the embedded Memory (eMEM), is programmed by the host system with the CIM kernel to execute and provides minimal support for stack and heap. The eCPU offloads vector instructions from the custom `xvnmc` ISA extension presented in Section III-B1 to the novel, area-efficient, scalable VPU described in Section III-B2. Unlike NM-Caesar, NM-Carus controller is Turing-complete and fully programmable using any RISC-V-compatible software build toolchain. To our knowledge, NM-Carus is the first proposal of a software-programmable, autonomous embedded memory device that supports vector operations through a newly proposed NMC-specific RISC-V ISA extension.

Despite integrating a CPU, NM-Carus remains a memory from a system perspective, following the same system integration model of NM-Caesar. As argued in Section II, the absence of integrated load-store units and master ports in its external interface is what differentiates NM-Carus from a traditional memory-mapped accelerator. On the other hand, the main difference between NM-Carus and an isolated vector coprocessor with a private memory is the emphasis on the memory capacity over processing resources. The VRF, which contributes to at least half of NM-Carus die area, is memory-mapped from a system perspective, and it is the only data source for the internal VPU. In contrast, traditional vector units load and store data from external memories, offering greater flexibility in VRF data layout at the cost of increased and more expensive data movements.

1) Instruction Set Architecture: The `xvnmc` RISC-V custom vector extension is specifically designed for small and efficient NMC devices to provide sufficient support for parallel workloads while limiting the hardware implementation cost (see Section IV-B). The extension is heavily inspired by the standard RISC-V Vector Extension (RVV) [39] and mostly shares the same instruction formats and semantics. RVV's built-in support for stripmining and very large vectors (up to 64 KiB vectors) perfectly aligns with NM-Carus scalable and memory-centric microarchitecture, allowing the same code to run on implementations with diverse VRF sizes.

The `xvnmc` custom instructions, listed in Table II, implement integer single-width arithmetic, logic, and shift opera-

tions as well as vector permutation instructions comparable to the standard RVV counterparts. Besides, `xvnmc` adds two instructions, `xvnmc.emvv` and `xvnmc.emvx`, dedicated to exchanging data between an arbitrary element of a vector register and an arbitrary scalar General-Purpose Register (GPR) in the eCPU. This is the only mechanism supported for the eCPU code to interact with the VRF content, which is not accessible through normal load and store instructions. Consequently, the eCPU instructions and private data are stored in the eMEM instead. Vector load and store instructions are not necessary: NM-Carus VPU works directly in the host system memory space, relying on the system to populate its VRF.

Table II: `xvnmc` instruction listing. Instruction variants are expanded in Table III. 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.v{mul,macc}[r]	vv	vx		Multiplication/MAC
xvnmc.v{and,or,xor}[r]	vv	vx	vi	Bitwise logic
xvnmc.v{min,max}[u][r]	vv	vx		Comparison
xvnmc.v{sll,srl,sra}[r]	vv	vx	vi	Logic/arith. shift
Vector Permutation Instructions				
xvnmc.vmv[r]	vv	vx	vi	Copy/splat into vector
xvnmc.vslide{up,down}[r]	vx	vi		Slide elements
xvnmc.vslide1{up,down}[r]	vx			Slide 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.

Table III: 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

One significant issue when combining a vector ISA without vector loads and stores with the proposed CIM model is that the layout of the data in the VRF is not chosen by the vector kernel. As a consequence, N vector instructions with different source and destination registers would be needed to iterate over N chunks, or rows, of the input data mapped to N vector registers. With standard vector instructions, this would translate into a full unrolling of the iteration process. Nested loops in the kernel exacerbate the problem, making the code size grow exponentially. This is not an issue in a traditional vector machine, which can conveniently load data from any memory location into the same vector registers in every loop iteration. In contrast, the data layout in NM-Carus VRF is dictated by the host system and thus potentially unknown at compile time if the memory is allocated dynamically. It would not be possible to use the same kernel for different register subsets if the source and destination registers were hardcoded as immediate values in the kernel's vector instructions as in the standard RVV. Moreover, the eMEM offers very limited space for the kernel code, whose size is, therefore, a major concern. To address both of these issues, the `xvnmc` extension provides instruction variants supporting *indirect register addressing*. These variants encode the index of the source and destination vector registers in the three least-significant bytes of a scalar GPR (`rs2`). This solution supports up to 256 logical vectors, accommodating applications with diverse data shapes, similar to [37]. As a result, the same vector instruction can be reused in every loop iteration with different operands by simply updating the GPR containing their indexes. Consequently, the code size is drastically reduced, and the same kernel can be used for input data with different base addresses and iteration counts (i.e., input data size). This mechanism is equivalent to updating the base address for a load or store instruction in a conventional software loop and, therefore, could easily be integrated into a custom compiler. Moreover, because a single GPR encodes all the instruction operands, updating their indexes only requires a single `add` instruction, further reducing the code size. The execution time of index-updating instructions is normally hidden by the latency of the last issued vector instruction, due to NM-Carus ability to execute scalar and vector instructions in parallel. The energy cost of index management instructions is negligible compared to vector operations, as shown by the low contribution of the eCPU to the total energy in Fig. 13. Finally, the absence of vector load and stores makes the `xvnmc` ISA completely independent of the data bitwidth, further increasing reusability compared to RVV.

In conclusion, the newly proposed `xvnmc` ISA offers a software-friendly, architecture-agnostic interface to leverage the proposed CIM model, bridging the host system's *memory* view of the CIM device with a convenient internal *register* view. Through vector processing, data-level parallelism and energy efficiency are increased, while a smaller code size and increased reusability compared to conventional vector ISAs are achieved using indirect vector register addressing.

From an application standpoint, the interaction between the host CPU and the NM-Carus instance is implemented through a driver that allows developers to program the eMEM inside the controller with a `xvnmc` program selected from a *library* of precompiled kernels. Although support for the `xvnmc` extension is currently available only at the assembler level, its compatibility with the standard RISC-V ISA and its similarity to the standard RVV facilitate potential integration at the compiler level for automatic kernel generation starting from high-level languages.

2) Microarchitecture: A block diagram of NM-Carus microarchitecture is presented in Fig. 4.

NM-Carus controller is essentially a minimal SoC featuring the OpenHW Group 4-stage, in-order CV32E40X [40]

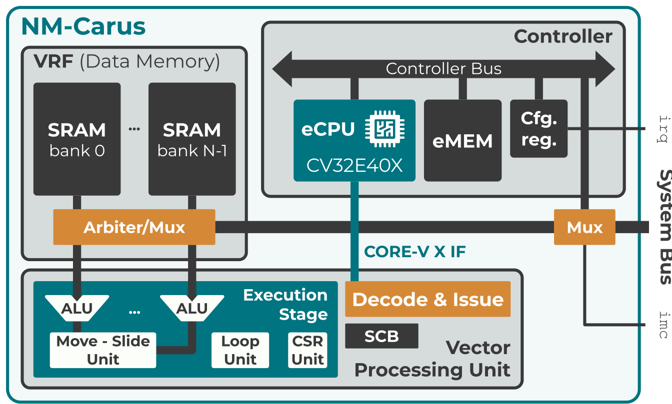


Figure 4: Top-level block diagram of NM-Carus.

RISC-V core[2] (eCPU), a tiny code memory (eMEM), and a configuration register that implements the synchronization interface with the host system. To minimize the core area, the eCPU implements the RV32EC ISA, offering 16 GPRs and no hardware multiplication and division. Instructions from the `xvnmc` extension are offloaded by the eCPU to the VPU through the CORE-V X interface [41]. All controller components are interconnected through a single-channel bus that is exposed to the host as a memory-mapped slave peripheral through the multiplexed memory-like interface that is also used to access NM-Carus data memory. In *configuration* mode, the host can program the eMEM with the kernel code or trigger and check the kernel execution by accessing the configuration register. NM-Carus can be set back to normal *memory* mode during the kernel execution so that normal memory operations are possible (e.g., to implement double buffering). Once the kernel terminates, a dedicated status bit is set to signal the end of the computation to the host system. As an alternative to software polling, this bit is routed to a dedicated optional pin in the NM-Carus top-level interface, which may be used as an interrupt source to allow the host system to enter a low-power state during computation. As explained in Section III-B1, `xvnmc.emvx` instructions are the only ones that cause data hazards between vector and scalar instructions. Therefore, in most cases, scalar and vector instructions can be executed in parallel as shown in Fig. 5, reducing the kernel execution time.

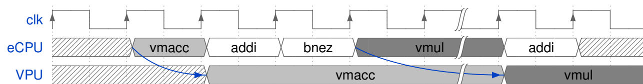


Figure 5: Scalar and vector instruction execution in NM-Carus.

NM-Carus VPU is implemented as a single-issue vector machine with a configurable level of hardware unrolling. The pipeline is composed of three main parts: (1) a *decode stage* interfacing with the eCPU and issuing decoded controls and scalar data to the execution units, (2) an *execution stage* with three dedicated units handling arithmetic, permutation, and CSR instructions, and (3) a shared *commit unit* keeping track of the execution status of the two in-flight instructions through a minimal scoreboard and handling synchronization with the eCPU. The entire VPU is clock-gated when no vector instructions are in flight. The rationale behind a single-issue design is to keep the control and arbitration logic and internal buffers as small as possible. Consequently, most of the available area and power budget is allocated to the arithmetic circuitry, which ultimately determines the throughput of the VPU. A wider or deeper execution pipeline typical of performance-oriented vector machines [11] would have guaranteed a greater utilization of the functional units and VRF data interfaces. However, because NM-Carus is intended to be configured with a large VRF, the idle time of the execution units is negligible compared to the total execution time of each instruction, which fails to justify the additional hardware and power cost required to minimize it. NM-Carus data memory is implemented as a configurable number of single-port, 32-bit SRAM banks with arbitrary capacity. The current VRF architecture supports 32 logical vector registers like the standard RVV. The chosen data interleaving policy, shown in Fig. 6, maps words that are contiguous inside the host system address space to adjacent VRF banks. All vector registers are naturally aligned with the physical banks, so elements with the same index from different vectors are physically mapped to the same VRF bank. This enables straightforward vector unrolling, with all the operands of an element-wise instruction being mapped to the same bank. Therefore, each ALU in the arithmetic unit is connected to a single SRAM bank, forming an independent computing *lane*. Consequently, NM-Carus VPU can be scaled arbitrarily: a higher number of lanes increases the unrolling level, thus improving throughput at the cost of increased area and power consumption. Because all operands come from the same bank, they must be accessed sequentially. This does not represent a bottleneck since NM-Carus relies on small serial arithmetic circuits, offering compelling advantages from an implementation point of view compared to parallel implementations. Providing the necessary throughput to parallel arithmetic units would also require larger and less efficient multi-port memory banks or more complex VRF layouts with additional routing and hazard-detection hardware. In addition, the supported vector instructions range from a minimum of one vector source operand (e.g., `xvnmc.vadd.vx`) per result to a maximum of three (e.g., `xvnmc.vmacc.vv`). Optimizing the memory access patterns for such a wide range of operations would exacerbate the issue.

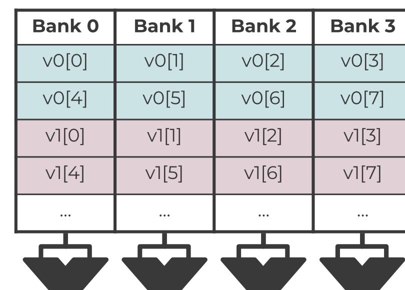


Figure 6: NM-Carus Vector Register File organization.

At the core of NM-Carus VPU is an execution engine with three dedicated execution units: (2.a) an *arithmetic unit* han-

[2]Smaller RISC-V CPUs exist and would better suit NM-Carus controller. CV32E40X was chosen because of its CORE-V X interface that allows straightforward implementation of custom coprocessors.

dling integer arithmetic and logic instructions, (2.b) a *move-slide unit* performing permutation and scalar-vector movement operations, and (2.c) a *CSR unit* managing the access to the `vtype` CSR encoding the current vector length and element bitwidth. A shared loop unit generates the VRF addresses to iterate among the elements of the source and destination vectors. The datapaths of (2.a) and (2.b) are interdependent since the ALUs inside the arithmetic unit and their internal registers are exploited to buffer intermediate data from both arithmetic and permutation operations. Similarly, resource sharing is heavily exploited inside the ALUs to implement packed-SIMD operations with minimal area. To this purpose, each ALU is equipped with a partitioned multi-precision 16-bit adder (similar to NM-Caesar's one), a 16-bit multiplier, elementary logic operators, and a serial 8-bit logic and arithmetic barrel shifter. 32-bit addition is performed as two successive 16-bit additions, 32-bit multiplication is computed with three successive 16-bit multiplications accumulated by the adder, and 8-bit multiplications are sign-extended to 16 bits and truncated. As a result, the adder can process 32-bit data words every two cycles regardless of the element bitwidth, while the multiplier produces four 8-bit, two 16-bit, and one 32-bit results in four, two, and three cycles respectively.

Every ALU performs the same operation on different elements of the same vector. Therefore, all lanes share the same ALU control unit and memory access scheduler. The compute and memory access patterns ensure that the throughput of the arithmetic unit is never lower than the slower unit between the ALU and the VRF for any combination of ALU operation and operand bitwidth. Most notably, an iteration of the `xvnmc.vmacc.vx` instruction at the core of many linear algebra operations is executed in each lane with a throughput of $1\,\mathrm{MAC/cycle}$, $0.67\,\mathrm{MAC/cycle}$, and $0.33\,\mathrm{MAC/cycle}$ for 8-bit, 16-bit, and 32-bit operands, respectively.

IV. Physical Implementation

Both NM-Caesar and NM-Carus have been implemented in a $32\,\mathrm{KiB}$ configuration on a low-power $65\,\mathrm{nm}$ CMOS technology library. The automated digital flow relies on Synopsys Design Compiler® 2020.09 for logic synthesis and Cadence Innovus® 20.1 for place and route. To meet the design goal of implementing two drop-in replacements for standard SRAM memories, the cycle time and input/output delay constraints under worst-case conditions were set to match those of a single-port reference $32\,\mathrm{KiB}$ memory macro generated by the same SRAM compiler used for internal SRAM banks. The same metal stack of standard SRAM macros was used, which favours little to no effort when integrating the NMC macros in a host system. The post-layout area and timing characteristics of the NMC macros are summarized in Table IV, while the post-synthesis area breakdown is shown in Fig. 7. The following sections discuss the implementation details of each NMC IC.

A. NM-Caesar

In a $2 \times 16\,\mathrm{KiB}$ configuration for its internal memories, NM-Caesar meets the target $330\,\mathrm{MHz}$ clock frequency of the reference $32\,\mathrm{KiB}$ SRAM bank. The output delay is unchanged, while the worst-case input delay is $2\,\%$ higher. The post-layout area is $673 \times 380\,\mathrm{\mu m}^2$, corresponding to an overhead of $28\,\%$ over the reference $32\,\mathrm{KiB}$ SRAM. Figure 8 shows the floorplan of NM-Caesar, highlighting its main internal components. The two memory banks are placed on the top of the design to minimize the distance between the logic and the interface pins in the lower part of the IC. This ensures the same pin orientation of the reference SRAM bank, easing the system-level routing phase while relaxing the timing constraints.

Table IV: Post-layout area and timing characteristics of a $32\,\mathrm{KiB}$ traditional SRAM, NM-Caesar, and NM-Carus when implemented on a low-power $65\,\mathrm{nm}$ CMOS technology node.

Metric	SRAM	NM-Caesar	NM-Carus
Post-layout area [$\mathrm{\mu m}^2$]	$200 \cdot 10^3$	$256 \cdot 10^3$ (+28 %)	$419 \cdot 10^3$ (+110 %)
Post-layout max clock freq. [MHz]	330	330	330
Max input delay [ns]	0.69	0.70 (+2 %)	0.70 (+2 %)
Max output delay [ns]	2.28	2.28 (+0 %)	2.48 (+9 %)

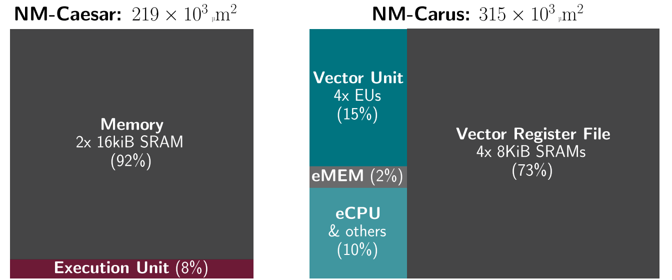


Figure 7: Post-synthesis area breakdown of $32\,\mathrm{KiB}$ instances of NM-Caesar and NM-Carus in accurate scale.

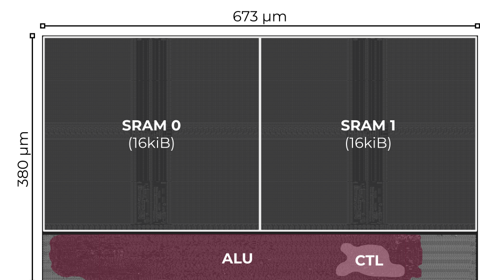


Figure 8: $32\,\mathrm{KiB}$ NM-Caesar layout.

B. NM-Carus

NM-Carus has been implemented in a four-lane configuration, with the controller and VPU placed in the middle of four $8\,\mathrm{KiB}$ SRAM macros constituting the internal VRF as shown in Fig. 9. The eMEM is implemented as a $512\,\mathrm{B}$ register file macro. The target $330\,\mathrm{MHz}$ clock frequency is met while the input and output delays are increased by $2\,\%$ and $9\,\%$ respectively due to the additional multiplexing logic that connects either the correct VRF bank or the controller bus to the shared external bus interface. The post-layout area is $705.0 \times 594.4\,\mathrm{\mu m}^2$, approximately corresponding to twice

the area of the reference 32 KiB SRAM, therefore meeting the target 50 % memory to logic ratio. Although most of the increased area overhead compared to NM-Caesar comes from NM-Carus additional logic, it is partially determined by the sublinear scaling of the footprint of an SRAM with its reduction in size. As visible in Fig. 7, NM-Carus 4-bank data memory is larger than NM-Caesar's 2-bank one, despite their identical capacity.

Placing the VRF banks at the corner of the NM-Carus floorplan partially counteracts the routing congestion, which is the main reason behind the low 60 % logic density. With this solution, the VPU vector permutation unit, which interconnects all the VRF banks, is placed in the center of the IC, minimizing the wire length. Extra space is left between adjacent memories to route the NM-Carus input and output pins to the middle logic part, limiting timing penalties while keeping the pins on the bottom side as in the reference macro. Implementations based on more advanced technology nodes would benefit from the higher scaling of logic compared to the SRAM arrays, reducing the area overhead or allowing for a higher lane count for a given area budget.

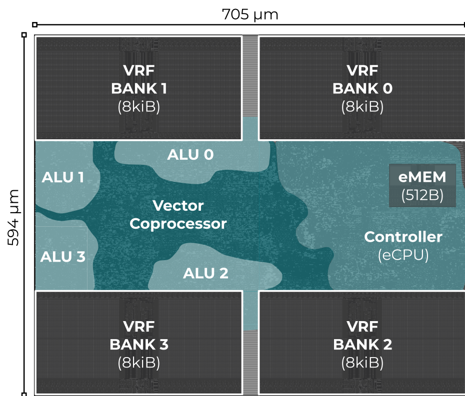


Figure 9: 32 KiB NM-Carus layout (4 lanes).

V. Performance and Energy Assessment

A. Experimental Setup

1) System Integration: The two 32 KiB NMC macros presented in Section IV were integrated inside X-HEEP [42], an open-source, configurable and low-power RISC-V MCU. Two of the eight 32 KiB SRAM banks connected to the X-HEEP bus were replaced by NM-Caesar and NM-Carus, and the necessary configuration registers were added to the peripheral subsystem of X-HEEP. NM-Carus interrupt pin was routed to the system CPU, the OpenHW Group CV32E40P in-order 4-stages RISC-V processor implementing the RV32IMC extensions. The necessary input data is directly embedded in the firmware and loaded into the NMC macros 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 on a low-power 65 nm technology library. All energy data presented in this section was collected from power analysis performed in Synopsys Primepower® 2019.12 under typical operating conditions using Value Change Dumps (VCDs) from system-level post-layout simulations when running selected benchmark applications.

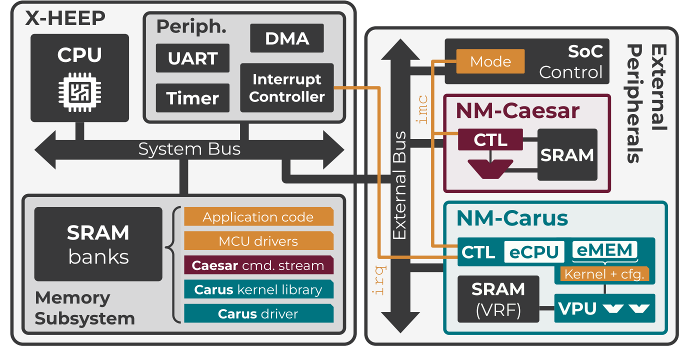


Figure 10: Block diagram and code placement of the HEEPerator system used to evaluate NM-Caesar and NM-Carus performance.

2) Benchmarks: To assess the system-level execution time and energy efficiency gains unlocked by the proposed NMC architectures, a set of representative computing kernels was selected and implemented in a CPU-only version used as a baseline and two NMC-enhanced variants for NM-Caesar and NM-Carus. The chosen kernels, listed in Table V and Fig. 11, constitute the basic building block 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-Caesar instruction sequences are generated using a dedicated in-house domain-specific compiler and streamed from memory to the NMC macro by the system DMA controller. NM-Carus kernels are written in C or RISC-V assembly, compiled using an extended version of the GNU RISC-V GCC compiler with assembler support for the `xvnmc` vector extension, and copied from memory to the eMEM at run time. Finally, the CPU-only kernels are compiled for the low-power system CPU using the RISC-V RV32IMC instruction set. To ensure consistency when comparing the cycle count, all the code is compiled with the `-O3` optimization flag, using the same GNU RISC-V GCC compiler, version 11.1.0.

B. Experimental Results

1) Recurrent kernels: Table V reports the relative execution time and energy reduction of the NMC-enhanced benchmark kernels compared to their reference CPU-only implementation. The speedup in arithmetic and linear algebra kernels ranges from 3.3 × to 28.0 × for NM-Caesar and from 6.6 × to 53.9 × for NM-Carus, depending on the data width. Simpler operations, such as bitwise XOR and element-wise addition, show no significant improvement when switching to lower-precision 16- and 8-bit data types. This is because the compiler's auto-vectorization optimizes the CPU code by packing multiple data items into a single GPR, resulting in a linear performance increase as data width decreases—similar to what occurs in the NMC devices. The execution time is further reduced up to 99.6 times in the case of Rectified Linear Unit (ReLU) in NM-Carus, thanks to the inclusion of minimum and maximum selection instructions in its `xvnmc` ISA, which brings a significant speedup compared to the CPU implementation relying on

Table V: System-level throughput and energy improvement of the HEEPerator system when executing the same kernels on NM-Caesar and NM-Carus, relative to their CPU-only implementation (baseline). Improvements are computed as CPU data divided by NM-Caesar or 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	2.5	61 pJ	4.0	99 pJ	11.0	267 pJ	112.0	2.88 nJ	73.1	1.91 nJ
	16-bit	5.0	124 pJ	11.0	269 pJ	11.0	285 pJ	112.0	3.00 nJ	81.2	2.26 nJ
	32-bit	10.0	281 pJ	10.0	278 pJ	10.0	279 pJ	89.1	2.54 nJ	66.3	1.95 nJ
Improvement		**Throughput**	**Energy**	**Throughput**	**Energy**	**Throughput**	**Energy**	**Throughput**	**Energy**	**Throughput**	**Energy**
NM-Caesar (32 KiB)	8-bit	5.0 ×	4.0 ×	8.0 ×	6.4 ×	22.0 ×	17.4 ×	**28.0** ×	**25.0** ×	**9.1** ×	**8.1** ×
	16-bit	5.0 ×	4.1 ×	11.0 ×	8.9 ×	11.0 ×	9.5 ×	14.0 ×	13.4 ×	6.7 ×	6.5 ×
	32-bit	5.0 ×	4.7 ×	5.0 ×	4.7 ×	5.0 ×	4.7 ×	5.6 ×	5.8 ×	3.3 ×	3.4 ×
NM-Carus (32 KiB, 4 lanes)	8-bit	12.7 ×	6.6 ×	20.3 ×	10.6 ×	42.0 ×	23.7 ×	**53.9** ×	**35.6** ×	**31.6** ×	**20.7** ×
	16-bit	12.7 ×	6.7 ×	27.9 ×	14.5 ×	27.9 ×	14.9 ×	37.1 ×	21.8 ×	24.1 ×	14.4 ×
	32-bit	12.7 ×	7.5 ×	12.7 ×	7.5 ×	12.6 ×	7.1 ×	11.0 ×	7.1 ×	7.3 ×	4.8 ×

[a] Input data size: 8 KiB (NM-Caesar), 10 KiB (CPU and NM-Carus). [b] $A[8,8] \times B[8,p]$, with $p = \{128, 256, 512\}$ (NM-Caesar) and $p = \{256, 512, 1024\}$ (CPU and NM-Carus) for $\{32, 16, 8\}$ bits.
[c] $\alpha\,(A[8,8] \times B[8,p]) + \beta C[8,p]$, with $p = \{128, 256, 512\}$ (NM-Caesar) and $p = \{256, 512, 1024\}$ (CPU and NM-Carus) for $\{32, 16, 8\}$ bits.

Device and bitwidth		2D convolution[d]		ReLU[e]		Leaky ReLU[e,f]		Maxpooling[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	344 pJ	12.0	300 pJ	64.6	1.44 nJ
	16-bit	133.0	3.4 nJ	12.0	338 pJ	11.5	295 pJ	65.6	1.5 nJ
	32-bit	115.1	3.1 nJ	10.0	300 pJ	9.5	258 pJ	50.3	1.2 nJ
Improvement		**Throughput**	**Energy**	**Throughput**	**Energy**	**Throughput**	**Energy**	**Throughput**	**Energy**
NM-Caesar (32 KiB)	8-bit	**16.9** ×	**14.2** ×	**26.0** ×	**22.4** ×	12.0 ×	10.3 ×	3.9 ×	3.8 ×
	16-bit	8.3 ×	7.6 ×	12.0 ×	11.6 ×	5.7 ×	5.0 ×	3.5 ×	3.5 ×
	32-bit	6.4 ×	6.1 ×	5.0 ×	5.1 ×	2.4 ×	2.2 ×	6.1 ×	5.8 ×
NM-Carus (32 KiB, 4 lanes)	8-bit	**47.5** ×	**29.4** ×	**99.6** ×	**59.3** ×	26.9 ×	17.3 ×	6.3 ×	6.7 ×
	16-bit	29.3 ×	17.6 ×	46.0 ×	28.9 ×	12.9 ×	8.6 ×	5.7 ×	5.8 ×
	32-bit	10.0 ×	6.3 ×	19.1 ×	2.8 ×	5.3 ×	3.7 ×	3.7 ×	3.5 ×

[d] $A[8,n] \circledast F[f,f]$, with $n = \{64, 64, 128\}$, $f = \{3, 4, 4\}$ (NM-Caesar) and $n = \{256, 512, 1024\}$, $f = 3$ (CPU and NM-Carus) for $\{32, 16, 8\}$ bits.
[e] Input data size: 8 KiB (NM-Caesar), 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: 8 KiB (NM-Caesar), 16 KiB (CPU and NM-Carus). NM-Caesar version partially implemented on the CPU.

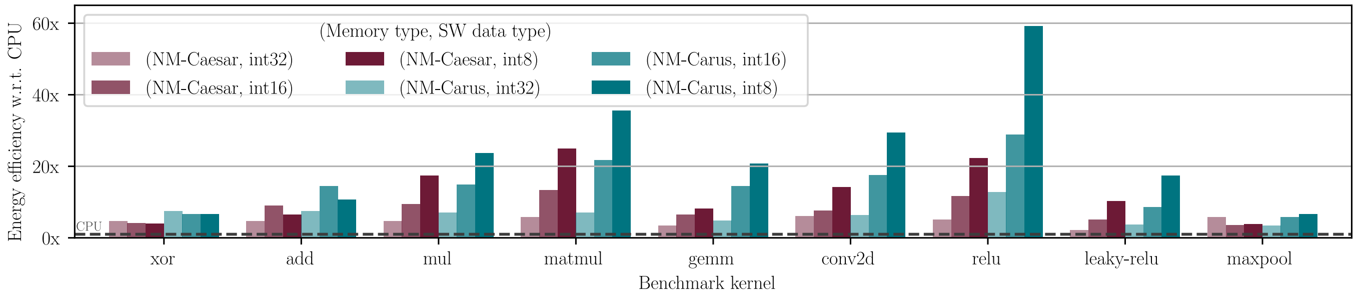
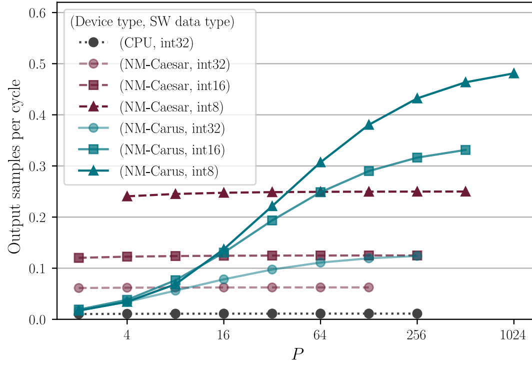


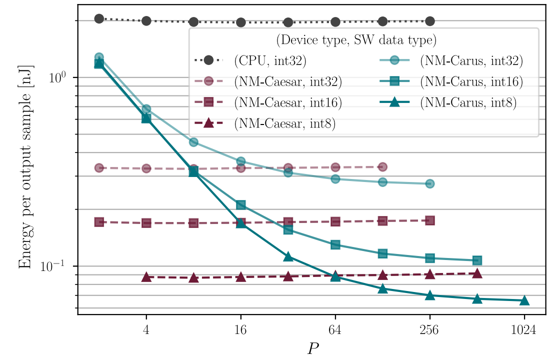
Figure 11: Energy efficiency gain of the NMC-enhanced HEEPerator MCU compared with its CPU-only version. Data source: post-layout simulation (65 nm, $f_{CLK} = 250$ MHz).

inefficient data-dependent branches instead. On the other hand, the NMC architectures only achieve marginal improvement in the execution time in the max pooling kernel. The lack of sub-word reduction operations in NM-Caesar and vector reduction operations in NM-Carus requires horizontal pooling to be implemented in software in the system CPU and on NM-Carus eCPU. In general, the energy efficiency of the proposed NMC ICs follows the same trend as the throughput, with significant gains over the CPU baseline, as better visualized in Fig. 11. It must be noted that the performance gain when moving to sub-word data sizes is uniquely due to the packed-SIMD architecture of the arithmetic units inside the NMC macros. The same approach has also been proven to be effective in conventional computing systems, typically in the form of dedicated ISA extensions such as [38].

The data in Table V was collected during kernel runs over large input datasets to show the maximum potential of the proposed architectures. However, to better demonstrate and motivate the architectural differences between NM-Caesar and NM-Carus, their performance was also measured when executing a matrix multiplication kernel on input matrices with varying dimensions. The results of this analysis, reported in Fig. 12, highlight two significantly different scaling trends. NM-Caesar throughput (Fig. 12a) and energy efficiency (Fig. 12b) improvements over the CPU baseline remain constant with the input size. Offloading workloads to NM-Caesar has a negligible overhead of five cycles, making it effective even with short computing tasks. In contrast, the overhead of NM-Carus CPU-based controller hinders its performance with small workloads, because the eCPU always executes a small sequence of instructions to bootstrap the offloaded kernel. Besides, as pointed out in Section III-B2, its single-issue, in-order VPU cannot guarantee optimal utilization of its ALUs between consecutive instructions since a limited number of ALU idle cycles is experienced while waiting for the writeback of the previous vector instruction to complete. Such overhead becomes negligible for larger data sizes, allowing NM-Carus to offer higher throughput and energy efficiency than NM-Caesar, saturating at $0.48\,\mathrm{output/cycle}$ over $0.25\,\mathrm{output/cycle}$ and $66\,\mathrm{pJ/output}$ over $175\,\mathrm{pJ/output}$ respectively with 8-bit



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



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

Figure 12: Throughput (a) and energy (b) scaling of NM-Caesar (red) and NM-Carus (cyan) compared to the CPU-only MCU version (grey) on several matrix multiplication kernels with different shape and bitwidth. The CPU throughput does not vary significantly with data size when using the standard RV32IMC ISA, therefore only 32-bit data is shown. Post-layout simulation data. Driver overhead not considered.

data. The significant difference in performance scaling further motivates the need for different architecture variants. Physical constraints and the nature of the expected workload are the primary factors determining which architecture best suits a given scenario. Ultimately, the choice depends on both application-specific requirements and system performance targets set by the designer. The low power consumption, minimal area overhead, and high bitcell density of NM-Caesar make it well-suited for ultra-low-power systems, where area and power are more critical than performance. Conversely, when performance and energy efficiency take precedence over area, the additional versatility and parallelism provided by NM-Carus offer greater benefits. Additionally, NM-Carus operates autonomously, freeing up the CPU for other tasks.

The results in Table V offer another important insight: the energy reduction of the NMC-enhanced kernels is lower than their throughput gain. In other words, the system draws more power when executing the same workload on the NMC macros compared to the CPU baseline. To better analyze power contributions in the system, a breakdown of the average power consumption during the execution of 8-bit and 32-bit 2D convolution kernels is shown in Fig. 13. The first observation is that memory accesses represent one of the largest contributions to system power consumption in all cases, supporting CIM as a viable solution to make edge computing more energy efficient. In the CPU case, memory accesses consume approximately as much power as the CPU itself. The NM-Caesar case shows a similar overall power consumption, where memory accesses account for almost 70 % of the total power, half of which is used to fetch the kernel micro-instructions and destination addresses from the system memory. Despite this, we have previously shown that NM-Caesar can process significantly more data than the CPU with the same power. The CPU disadvantage is primarily due to the explicit instructions required to move the input data between the system memory and the CPU GPRs, which are not necessary in the NMC architectures. In addition, the NM-Caesar power breakdown highlights the potential benefits of a dedicated controller that generates instructions on-the-fly instead of reading them from the system memory. This would allow NM-Caesar to achieve the same level of autonomy as NM-Carus and possibly a similar energy efficiency at the cost of reduced flexibility. Finally, NM-Carus case confirms the advantages of exploiting a vector-capable architecture that makes the power contribution of its CPU-based controller negligible compared the accesses to its four VRF SRAM banks (twice as NM-Caesar's), which account for 60 % of the total system power, and significantly more than its computing resources, that consume less power than the system CPU despite producing far more output samples in the same amount of time.

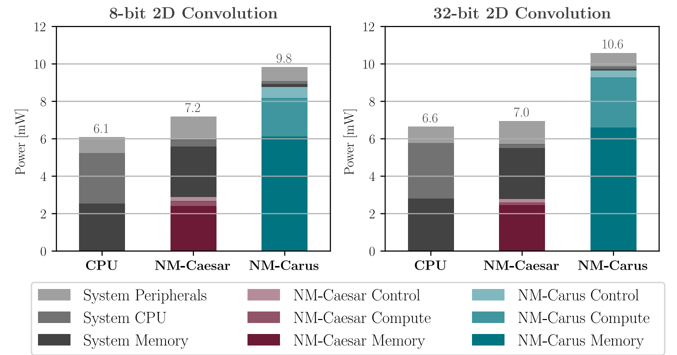


Figure 13: Average power breakdown in CPU (grey), NM-Caesar (red), and NM-Carus (cyan) for 8-bit and 32-bit 2D convolution. Post-layout simulation data (65 nm, $f_{CLK} = 250\,\text{MHz}$)
.

2) Anomaly detection end-to-end application: To better assess the performance of NM-Caesar and NM-Carus on a representative workload, the full *Anomaly Detection* TinyML algorithm [43] is deployed on the HEEPerator testbench system (Fig. 10) and compared against a multi-core CPU cluster featuring the CV32E40P RISC-V core [38] with its DSP-enhanced RV32IMCXcv ISA. The application implements an autoencoder composed of ten matrix-vector multiplication layers with ReLU activation functions. Table VI reports the cycle count, energy consumption, and area of a minimal system configured with a single-, dual-, and quad-core CV32E40P, as well as a tiny RISC-V CV32E20 core (referred to as micro-riscy in [44]) coupled with either NM-Caesar or NM-Carus. The CPU-based configurations include a single 32 KiB SRAM bank as

the L1 data cache, which is replaced with the NMC devices in the respective configurations. To ensure a fair comparison, the multi-core CPU configurations are evaluated under the assumption of ideal linear scaling in both performance and area. Additionally, to focus the comparison on the computational capabilities of the systems, the contribution of the instruction memory is excluded from the energy consumption analysis. As highlighted in the table, NM-Caesar achieves a 30 % higher throughput and 20 % lower energy consumption compared to the single-core system, while also occupying 10 % less area. However, it is slower and less energy-efficient than the multi-core configurations, making it an ideal candidate for area-constrained devices. On the other hand, NM-Carus performs within 13 % of the quad-core system while consuming half the energy and requiring less area than the dual-core configuration. This is a significant result, as it demonstrates the scalability potential of a multi-instance NM-Carus-based data memory subsystem for energy-constrained applications.

Table VI: Execution time and energy improvements and area overhead of NM-Caesar, NM-Carus, and two multi-core CPU-based systems (dual- and quad-core) with respect to a single-core CPU-based system when running the *Anomaly Detection* TinyML application.

Metric	**CV32E40P**[a] (1 core)	**CV32E40P**[a] (2 cores)	**CV32E40P**[a] (4 cores)	**NM-Caesar + CV32E20**[b]	**NM-Carus + CV32E20**[b]
Cycle count	$561 \cdot 10^3$	↓ 2.00 ×	**↓ 4.00 ×**	↓ 1.29 ×	↓ 3.55 ×
Energy[c] [μJ]	13.5	↓ 1.37 ×	↓ 1.67 ×	↓ 1.20 ×	**↓ 2.36 ×**
Post-layout area [μm^2]	$350 \cdot 10^3$	↑ 1.43 ×	↑ 2.29 ×	**↓ 0.90 ×**	↑ 1.36 ×

[a] With RV32IMCXcv ISA. [b] With RV32E ISA. [c] At 250 MHz; post-layout data.

C. Comparison with the State of the Art

To better position the proposed NM-Caesar and NM-Carus NMC architectures in the context of state-of-the-art CIM solutions, we compare them against recent representative IMC and NMC ICs: (1) BLADE [35], an in-SRAM computing architecture that utilizes local wordline groups to perform energy–efficient computations, (2) Computational SRAM (C-SRAM) [34], a hybrid IMC and NMC scalable integrated vector unit, and (3) Vecim [10], a high-performance vector coprocessor based on [11] that exploits IMC to offer top-in-class energy efficiency. Table VII provides a quantitative and qualitative overview of the features, architecture, physical characteristics, and performance of these designs compared to NM-Caesar and NM-Carus despite their technology, size and architecture differences. To establish a fair comparison from a terchnological standpoint among the different works, BLADE's and C-SRAM's physical characteristics 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 at the basis on NM-Caesar, NM-Carus, and Vecim. Since both designs are memory-dominated, the scaling factors were applied to both the memory and logic area, thus resulting in optimistic 65 nm values for BLADE and C-SRAM. The energy scaling factors have been 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 the original 28 nm (BLADE) or 22 nm (C-SRAM) nodes.

Among the considered architectures, NM-Carus achieves the highest peak energy efficiency, that is 6 % higher than Vecim and 21 % higher than the multi-array BLADE implementation. NM-Caesar excels in memory density while maintaining high energy efficiency, especially if coupled with a dedicated controller mentioned previously. The Vecim vector coprocessor delivers the best peak performance thanks to its highly parallel execution units while offering a throughput per area within 15 % to the multi-array BLADE instance. A similar performance density is expected from NM-Carus instances with a higher lane count, since its throughput scales almost linearly with the number of ALUs, while the area overhead of the additional logic and the smaller VRF banks is contained.

To better compare the performance of the considered CIM architectures, Table VIII reports the peak throughput and the energy consumption of BLADE, C-SRAM, NM-Caesar, and NM-Carus when executing a matrix multiplication on different data widths. The throughput of BLADE and C-SRAM was estimated based on the values reported in the respective articles for multiplication operations. Because they have been implemented as 2 KiB and 4 KiB macros, we considered 16 and 8 instances, respectively, to match the 32 KiB memory capacity of NM-Caesar and NM-Carus. A single 32 KiB instance of BLADE is also included to show its scaling versus capacity. Structural hazards and data replication for data sizes exceeding the array's dimension are not considered for BLADE and C-SRAM. Similarly, we only consider the proportional increase in static leakage power for a 32 KiB BLADE subarray when evaluating its energy consumption, not taking into account the increase in dynamic power. All these choices were made not to penalize or underestimate the performance of BLADE and C-SRAM, placing them under optimal conditions in an attempt to provide a fair comparison with the NMC architectures proposed in this work despite the different technology nodes. NM-Carus achieves 1.31 × and 1.97 × faster execution time than the multi-array BLADE implementation on 16-bit and 8-bit data respectively, performing 2.1 × worse only in the 8-bit kernel because of the better scaling of BLADE's add-and-shift multiplier architecture for smaller data types. However, BLADE's performance may be degraded in real-world scenarios because of unsupported inter-bank operations. NM-Carus is also the most energy-efficient design, consuming up to 3 × less energy than BLADE on 32-bit data. NM-Caesar, on the other hand, achieves the same 32-bit performance of BLADE at 60 % its power consumption and half its area. The single-array BLADE instance and the multi-array C-SRAM perform worse than NM-Carus or the multi-array BLADE in all cases. In this regard, it must be noted that the C-SRAM data has been extracted from silicon measurements instead of post-layout simulations, possibly explaining the discrepancy.

With the increasing deployment of computationally intensive algorithms on low-power devices, reduced-precision models utilizing sub-byte data types are becoming increasingly popular. CIM architectures that employ serial multipliers, such as the *add-and-shift* units in BLADE and C-SRAM, would achieve higher energy efficiency than the proposed NMC

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

	BLADE [35]		**C-SRAM** [34], [45]		**Vecim** [10]	**NM-Caesar** (this work)	**NM-Carus** (this work)
CIM type	IMC		IMC + NMC		IMC + NMC	NMC	NMC
Array instances	16 × 2 KiB		4 × 8 KiB		1 × 16 KiB (4 lanes)	1 × 32 KiB	1 × 32 KiB (4 lanes)
SRAM type	Foundry 6T, custom array		Foundry 8T, custom array		Foundry 8T, custom array	Foundry 6T	Foundry 6T
Usefull bitcell density[a] [%]	**53.5**		20.3		1.7	**54.0**	33.0
Deployment constraints	• Word alignment • Data placement (LG)		• Word alignment • Data replication ($D, \overline{D}$)		• Vector alignment	• Word alignment	• Vector alignment
Instance ALU features	32-bit SIMD ALU		128-bit SIMD ALU		4 × 256-bit SIMD ALUs	2-cycle 32-bit SIMD ALU	4 × 32-bit SIMD ALUs
Multiplier architecture	1-bit Add and shift		Add and shift		Double-rate bit-parall.	4 × 17-bit multipliers	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)	• ADD/SUB • MULT • MAC • Logical shift • Logic • Comparison	• 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	65 nm
Area [μm²]	$64 \cdot 10^3$	$580 \cdot 10^3$	$17.5 \cdot 10^3$	N/A[c]	$4 \cdot 10^6$	$256 \cdot 10^3$	$415 \cdot 10^3$
Nominal freq. [MHz]	2200	330[d]	1000	330[d]	250	330	330
Peak throughput[e] [GOPS]	**35.2**	5.3	10.7	3.5	**31.8**	1.32	2.64
Energy eff. [GOPS/W]	830.7[f]	254.2[f]	52.0	13.2	289.1	200.3 (421.9[f])[f]	**306.7**[g]
Area eff. [GOPS/mm²]	550[f]	**9.1**	611	N/A[c]	8.0	5.2	6.4

[a] Area of useful bitcells (i.e., replicated data is not considered), normalized to 6T bitcell area. [b] Area, timing, and power at 65 nm were computed from 28 nm ones based on commercial SRAM scaling on the same technologies. This is a conservative, best-case scaling factor for the standard-cell parts of the design. [c] Area estimation from 22 nm data is not trivial because of C-SRAM's mixed IMC/NMC design, so it's not provided. [d] Assumed to match the operating frequency of the 65 nm 32 KiB SRAM used as reference to define NM-Caesar and 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. [g] Average power from post-layout simulation while running a matrix multiplication kernel.

Table VIII: Peak performance comparison on matrix multiplications.

Metric and bitwidth		**BLADE**[a,b] 16 × 2 KiB 28 nm	**BLADE**[a,b] 16 × 2 KiB 65 nm	**BLADE** 1 × 32 KiB 28 nm	**BLADE** 1 × 32 KiB 65 nm	**C-SRAM**[a,b] 8 ×4 KiB 22 nm	**C-SRAM**[a,b] 8 ×4 KiB 65 nm	**NM-Caesar** (this work) 65 nm	**NM-Carus** (this work) 65 nm
Cycle count	8-bit[d]	$\mathbf{12.8 \cdot 10^3}$		$204.8 \cdot 10^3$		$19.2 \cdot 10^3$		$51.2 \cdot 10^3$	$26.6 \cdot 10^3$
	16-bit[e]	$25.6 \cdot 10^3$		$409.6 \cdot 10^3$		$38.4 \cdot 10^3$		$51.2 \cdot 10^3$	$\mathbf{19.5 \cdot 10^3}$
	32-bit[f]	$51.2 \cdot 10^3$		$819.2 \cdot 10^3$		$76.8 \cdot 10^3$		$51.2 \cdot 10^3$	$\mathbf{26.0 \cdot 10^3}$
Execution time[c] [μs]	8-bit[d]	5.8	**38.8**	93	620	19.3	58.1	155	80.6
	16-bit[e]	11.6	77.5	186	1240	38.4	116	155	**59.1**
	32-bit[f]	23.3	155	372	2480	76.8	232	155	**78.8**
Energy [pJ/MAC]	8-bit[d]	2.4	7.9	13	43	38.8	150	16.3	**6.8**
	16-bit[e]	8.1	26.7	29.4	97.1	155	600	32	**12.0**
	32-bit[f]	31.1	103	96.9	320	621	2400	61.8	**31.2**

[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] $A[10, 10] \times B[10, 256]$

devices when dealing with such models. However, as discussed in Section III, the ALUs of NM-Caesar and NM-Carus were specifically designed for standard data types,which gives no advantage when handling lower-precision models, as their data would be sign-extended to 8 bits.

VI. Conclusion

This article has presented NM-Caesar and NM-Carus, two novel architectural variations of a new NMC approach that targets energy-efficient and low-power SoC designs for edge computing devices running TinyML algorithms on the edge. Leveraging the NMC paradigm, they enable the execution of arithmetic operations next to the memory array without the need to transfer data through the system bus. By performing computation on vectors in a SIMD manner, they can improve the throughput and energy performance of conventional architecture relying on energy-efficient CPUs or state-of-the-art IMC/NMC architectures. Experimental results have shown that NM-Caesar and NM-Carus achieve a timing speed-up of up to $25.8\times$ and $50.0\times$, as well as a reduction in energy of $23.2\times$ and $33.1\times$ compared to a scalar architecture based on RISC-V in a matrix multiplication kernel. In particular, a 65 nm implementation of NM-Carus demonstrates a peak energy efficiency of 306.7 GOPS/W with 8-bit data. In addition, their software-friendly digital-based design approach will ease their integration into existing systems without any major changes for optimal performance across different technology nodes.

VII. Acknowledgements

This work was supported in part by the Swiss State Secretariat for Education, Research, and Innovation (SERI) through the SwissChips research project.

References

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

[Online]. Available: https://www.biorxiv.org/content/early/2024/02/28/2024.02.23.581779

[2] 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.

[3] J. Backus, "Can programming be liberated from the von neumann style? A functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, aug 1978.

[4] 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, iSSN: 2376-8606.

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

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

[7] S. Srinivasa, A. K. Ramanathan, J. Sundaram, D. Kurian, S. Gopal, N. Jain, A. Srinivasan, R. Iyer, V. Narayanan, and T. Karnik, "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.

[8] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "Near-memory computing: Past, present, and future," *Microprocessors and Microsystems*, vol. 71, p. 102868, 2019.

[9] 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. New York, NY, USA: Association for Computing Machinery, 2019.

[10] 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.

[11] 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.

[12] E. De Giovanni, F. Montagna, B. W. Denkinger, S. Machetti, M. Peón-Quirós, S. Benatti, D. Rossi, L. Benini, and D. Atienza, "Modular design and optimization of biomedical applications for ultralow power heterogeneous platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3821–3832, 2020.

[13] Z. He, X. Zhang, Y. Cao, Z. Liu, B. Zhang, and X. Wang, "LiteNet: Lightweight neural network for detecting arrhythmias at resource-constrained mobile devices," *Sensors*, vol. 18, no. 4, p. 1229, 2018.

[14] S. Soro, "TinyML for ubiquitous edge AI," 2021.

[15] C. Gómez, P. Arbeláez, M. Navarrete, C. Alvarado-Rojas, M. Le Van Quyen, and M. Valderrama, "Automatic seizure detection based on imaged-EEG signals through fully convolutional networks," *Scientific reports*, vol. 10, no. 1, p. 21833, 2020.

[16] U. R. Acharya, H. Fujita, O. S. Lih, Y. Hagiwara, J. H. Tan, and M. Adam, "Automated detection of arrhythmias using different intervals of tachycardia ECG segments with convolutional neural network," *Information sciences*, vol. 405, pp. 81–90, 2017.

[17] 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.

[18] F. Conti, D. Rossi, G. Paulin, A. Garofalo, A. Di Mauro, G. Rutishauer, G. m. Ottavi, M. Eggimann, H. Okuhara, V. Huard, O. Montfort, L. Jure, N. Exibard, P. Gouedo, M. Louvat, E. Botte, and L. Benini, "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.

[19] C. Xin, Q. Chen, M. Tian, M. Ji, C. Zou, X. Wang, and B. Wang, "COSY: An energy-efficient hardware architecture for deep convolutional neural networks based on systolic array," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, 2017, pp. 180–189.

[20] C. Zhang, X. Wang, S. Yong, Y. Zhang, Q. Li, and C. Wang, "An energy-efficient convolutional neural network processor architecture based on a systolic array," *Applied Sciences*, vol. 12, no. 24, p. 12633, 2022.

[21] M. Sinigaglia, L. Bertaccini, L. Valente, A. Garofalo, S. Benatti, L. Benini, F. Conti, and D. Rossi, "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.

[22] 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*, ser. CF '23, 2023, pp. 381–382.

[23] P. D. Schiavone, D. Rossi, A. Di Mauro, F. K. Gürkaynak, T. Saxe, M. Wang, K. C. Yap, and L. Benini, "Arnold: An eFPGA-augmented RISC-V SoC for flexible and low-power IoT end nodes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 677–690, 2021.

[24] R. R. Álvarez, B. Denkinger, J. Sapriza, J. M. Calero, G. Ansaloni, and D. Atienza, "An open-hardware coarse-grained reconfigurable array for edge computing," in *Proceedings of the 20th ACM International Conference on Computing Frontiers*, ser. CF '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 391–392.

[25] Z. Ebrahimi and A. Kumar, "BioCare: An energy-efficient CGRA for bio-signal processing at the edge," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[26] 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.

[27] 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)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), B. B. Brandenburg, Ed., vol. 196. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 1:1–1:18.

[28] A. Pullini, D. Rossi, I. Loi, G. Tagliavini, and L. Benini, "Mr.Wolf: An energy-precision scalable parallel ultra low power SoC for IoT edge processing," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 7, pp. 1970–1981, 2019.

[29] 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.

[30] 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, 2023.

[31] C.-X. Xue, W.-H. Chen, J.-S. Liu, J.-F. Li, W.-Y. Lin, W.-E. Lin, J.-H. Wang, W.-C. Wei, T.-Y. Huang, T.-W. Chang, T.-C. Chang, H.-Y. Kao, Y.-C. Chiu, C.-Y. Lee, Y.-C. King, C.-J. Lin, R.-S. Liu, C.-C. Hsieh, K.-T. Tang, and M.-F. Chang, "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, 2020.

[32] W. Banerjee, "Challenges and applications of emerging nonvolatile memory devices," *Electronics*, vol. 9, no. 6, 2020.

[33] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable processors in the billion-transistor era: IRAM," *Computer*, pp. 75–78, 1997.

[34] M. Kooli, A. Heraud, H.-P. Charles, B. Giraud, R. Gauchi, M. Ezzadeen, K. Mambu, V. Egloff, and J.-P. Noel, "Towards a truly integrated vector processing unit for memory-bound applications based on a cost-competitive computational SRAM design solution," *J. Emerg. Technol. Comput. Syst.*, vol. 18, no. 2, apr 2022.

[35] 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.

[36] 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*, ser. ISPD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 43–46.

[37] R. Gauchi, V. Egloff, M. Kooli, J.-P. Noel, B. Giraud, P. Vivet, S. Mitra, and H.-P. Charles, "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. New York, NY, USA: Association for Computing Machinery, 2020, pp. 121–126.

[38] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable iot endpoint devices," *IEEE*

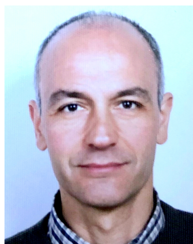
Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 10, pp. 2700–2713, 2017.
[39] RISC-V Corp., *RISC-V 'V' Vector Extension, version 1.0*, 2022. [Online]. Available: https://github.com/riscv/riscv-v-spec
[40] OpenHW Group, "OpenHW Group CV32E40X user manual," online, 2023, accessed: today. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/index.html
[41] ——, "OpenHW Group specification: Core-V eXtension interface (CV-X-IF)," online, 2023, accessed: today. [Online]. Available: https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/latest/index.html
[42] 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.
[43] Unknown, "Anomaly detection," online, 2020, accessed: today. [Online]. Available: https://github.com/mlcommons/tiny/tree/master/benchmark/training/anomaly_detection
[44] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications," pp. 1–8, 2017.
[45] J.-P. Noel, M. Pezzin, R. Gauchi, J.-F. Christmann, M. Kooli, H.-P. Charles, L. Ciampolini, M. Diallo, F. Lepin, B. Blampey *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.

Michele Caon is a postdoctoral researcher at the Embedded Systems Laboratory of EPFL. He received his B.S. and M.S., and Ph.D. at the Electronics and Telecommunications Department at Politecnico di Torino in 2017, 2019, and 2024. His research interests are innovative digital, integrated, programmable computing circuits and systems. He is currently working on near-memory computing circuits embedded in heterogeneous systems-on-chip for edge computing applications.

Clément Choné received the MSc degree in Micro and Nanotechnologies for integrated systems from Grenoble-INP Phelma, in 2022. He is currently working towards his PhD degree with the Embedded Systems Laboratory (ESL) of EPFL, Switzerland, under the supervision of Prof. David Atienza and Prof. Andreas Burg. His main research interests include the development of near-memory computing accelerators and low-power logic and memory systems using innovative circuit styles to suppress leakage current in low-power systems.

Pasquale Davide Schiavone is a postdoctoral researcher at the Embedded Systems Laboratory of EPFL and Director of Engineering of the OpenHW Group. He obtained the Ph.D. title at the Integrated Systems Laboratory of ETH Zurich in the Digital Systems group in 2020 and the BSc. and MSc. from "Politecnico di Torino" in computer engineering in 2013 and 2016, respectively. His main activities are the RISC-V CPU design and low-power energy-efficient computer architectures for smart embedded systems and edge-computing devices.

Alexandre Levisse received his Ph.D. degree in Electrical Engineering from CEA-LETI and Aix-Marseille University, France, in 2017. From 2018 to 2021, he was a postdoctoral researcher at the Embedded Systems Laboratory of EPFL. From 2021, he works as a scientist in EPFL. His research interests include circuits and architectures for emerging memory and transistor technologies as well as in-memory computing and accelerators.



Guido Masera (SM'07) received the Dr.-Ing. (summa cum laude) and Ph.D. degrees in Electronic Engineering from Politecnico di Torino, Italy. He has been a professor with the Electronics and Telecommunications Department, Politecnico di Torino, since 1992. His research interests focus digital integrated circuits and systems, with a special emphasis on high-performance architectures for communications, forward error correction, image and video coding, cryptography and hardware accelerators for machine learning. He has more than 200 publications, two patents and was a designer of several ASIC components. Dr. Masera is an Associate Editor of MDPI Electronics and a former Associate Editor of the IEEE Transactions on Circuits and Systems I, IEEE Transactions on Circuits and Systems II and the IET Circuits, Devices & Systems.

Maurizio Martina received the Dr.-Ing. and Ph.D. degrees in electronic engineering and electronic and communications engineering from Politecnico di Torino, Italy, in 2000 and 2004. He is a Professor with the Electronics and Telecommunications Department, Politecnico di Torino, since 2014. His research interests include computer architecture and VLSI design of digital integrated circuits for image and video coding, forward error correction, cryptography and artificial intelligence. He has more than 100 publications and holds two patents. He served as an Associate Editor of the IEEE Transactions on Circuits and Systems—I and as a Guest Editor of several special issues, including BioCAS 2017 special issue in IEEE Transactions of Biomedical Circuits and Systems and ISCAS 2023 special issue in IEEE Transactions on Circuits and Systems-II. He has been part of the organizing and technical committee of several IEEE conferences, including BioCAS 2017, AICAS 2020, and PRIME 2025.

David Atienza (M'05-SM'13-F'16) received his MSc and PhD degrees in Computer Science and Engineering from Complutense Univ. of Madrid, Spain, and IMEC, Belgium, in 2001 and 2005. He is a professor of Electrical and Computer Engineering, Associate Vice President of Research Centers and Platforms, and heads the Embedded Systems Laboratory (ESL) at EPFL, Switzerland. His research interests include system-level design methodologies for high-performance multi-processor system-on-chip and low-power Internet-of-Things systems, including new thermal-aware design for 2-D/3-D MPSoCs and many-core servers, ultra-low power system architectures for wearable systems and edge AI computing, and memory hierarchy optimizations. He is a co-author of more than 450 publications and 14 patents. He serves as Editor-in-Chief of IEEE TCAD (period 2022-2025) and ACM CSUR (since 2024). Dr. Atienza received the 2024 Test-of-Time Best Paper Award at CODES+ISSS, the ICCAD 10-Year Retrospective Most Influential Paper Award, and the DAC Under-40 Innovators Award, among others. He is a Fellow of the ACM.