# Bank on Compute-near-Memory: Design Space Exploration of Processing-near-Bank Architectures

Rafael Medina, Giovanni Ansaloni, Marina Zapater, Alexandre Levisse, Saeideh Alinezhad Chamazcoti, Timon Evenblij, Dwaipayan Biswas, Francky Catthoor, David Atienza

***Abstract*—Near-DRAM computing strategies advocate for providing computational capabilities close to where data is stored. Although this paradigm can effectively address the memory-to-processor communication bottleneck, it also presents new challenges: The strict resource constraints in the memory periphery demand careful tailoring of architectural elements. We herein propose a novel framework and methodology to explore Compute-near-Memory designs that interface to DRAM memory banks, demonstrating the area, energy, and performance trade-offs subject to the architectural configuration. We exemplify this methodology by conducting two studies on compute-near-bank designs: (1) analyzing the interaction between control and data resources, and (2) exploring the integration of processing units with different DRAM standards. According to our study, the optimal size ratios between instruction and data capacity vary from $2\times$ to $4\times$ across benchmarks from representative application domains. The retrieved Pareto-optimal solutions from our framework improve state-of-the-art designs, e.g., achieving a 50% performance increase on matrix operations with 15% energy overhead relative to the FIMDRAM design. In addition, the exploration of DRAM shows the interplay between available internal bandwidth, performance, and area overhead. For example, a three-fold increase in bandwidth rises performance by 47% across workloads at a 34% extra area cost.**

***Index Terms*—Compute-near-Memory, DRAM, Processing-in-Memory, Accelerator, System simulation, Performance evaluation**

## I. Introduction

Modern applications, particularly in the high performance computing, machine learning and data processing domains [1], have grown significantly in memory footprint and computational intensity. This trend poses a challenge for data transfers throughout the system, exacerbating the performance disparity between computation and memory, the so-called memory wall [2]. As a result, a significant part of execution time and energy is devoted to off-chip communication, stalling computation [3]–[7].

Near-data processing architectures alleviate these shortcomings by placing computing where the data is stored. These designs reduce the need for communication between main processors and memory elements, decreasing the latency in the system and increasing energy efficiency. In addition, their close access to memory allows high parallelism when executing repetitive kernels [8]–[10].

Near-data processing alternatives can be divided into Compute-in-Memory (CiM), where the array of memory cells is customized to enable computation between memory words, and Compute-near-Memory (CnM), which places processing units (PUs) close to the cell arrays without modifying them. Both strategies can be implemented at any point of the system memory hierarchy —caches, main memory, or storage— offering different degrees of parallelism [8], [9], [11]. Among these, CnM at the bank level is a particularly promising strategy [10], [12]–[17]. It involves closely interfacing the PUs with DRAM banks, avoiding (1) the costly modification of the cell array IPs, (2) the stringent area restrictions within the bank, and (3) the energy overhead to move data between bank and DRAM IO. By also leveraging simultaneous access to banks, bank-level CnM architectures offer high-throughput, low-latency, and low-energy data processing [9], [10].

Although several bank-level CnM designs have been proposed [10], [12]–[17], they highlight individual design points, without an analysis of architectural parameters, DRAM protocols, and their impact on performance. Therefore, they cannot provide general guidelines. Instead, we introduce a new methodology to systematically explore the Compute-near-Memory design space for diverse application domains, conforming to the DRAM protocol modifications introduced by this paradigm. By simulating the processing units and their interface to the DRAM banks, and providing a programming model, we can analyze the effect of a wide array of architectural choices on performance, energy, and area. To this end, we provide a template based on the state-of-the-art FIMDRAM architecture [16]. The template enables the parametric definition of CnM solutions that interface processing units with the DRAM memory banks, in compliance with available JEDEC standards. We demonstrate the versatility of this methodology through the analysis of two crucial dimensions of the CnM design, and quantify their trade-offs for the first time. First, we study the balance between control and data resources at the PU (i.e., the storage capacity for instructions and variables), which is essential under the CnM area constraints to further exploit data locality at the DRAM proximity. Second, we explore how interfacing PUs to banks in different DRAM standards impacts computing behavior, showcasing that the standard choice can target the optimization of different performance metrics to improve upon state-of-the-art designs. In summary, the contributions of this paper are the following:

- We introduce a Compute-near-Memory architectural template that allows to model PUs and the interfaced DRAM protocol according to selected architectural parameters. We also provide a design exploration framework and

Rafael Medina, Giovanni Ansaloni, Alexandre Levisse, and David Atienza are with the Embedded Systems Laboratory (ESL), EPFL, 1015 Lausanne, Switzerland (e-mail: {rafael.medinamorillas, giovanni.ansaloni, alexandre.levisse, david.atienza}@epfl.ch).

Marina Zapater is with the REDS institute, HEIG-VD, HES-SO, 1401 Yverdon-les-Bains, Switzerland (e-mail: marina.zapater@heig-vd.ch).

Saeideh Alinezhad Chamazcoti, Timon Evenblij, Dwaipayan Biswas, and Francky Catthoor are with IMEC, 3001 Leuven, Belgium (e-mail: {saeideh.alin, timon.evenblij, dwaipayan.biswas, francky.catthoor}@imec.be).

This work has been partially supported by a joint research grant for ESL-EPFL by IMEC, by the EC H2020 FVLLMONTI project (GA No. 101016776), by ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR, and by the Swiss State Secretariat for Education, Research, and Innovation (SERI) through the SwissChips research project.

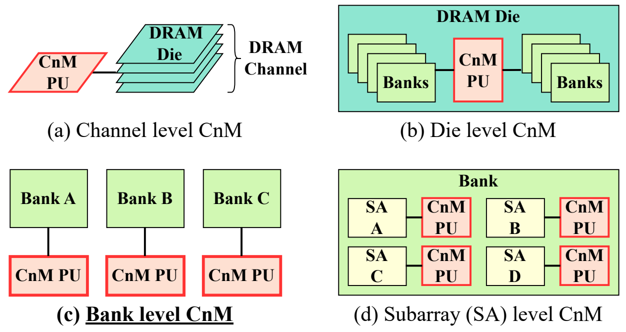


Fig. 1. Diagram of the different levels of the DRAM structure where Compute-near-Memory (CnM) Processing Units (PUs) can be interfaced.

a programming interface to simulate the execution of applications on individual instances of the template.

- We explore the trends in performance, area, and energy consumption of bank-level CnM PUs across design points, validated with ML and data processing compute kernels. For example, we show that FIMDRAM [16] can be outperformed by more than 1.95x, with only a 23% energy overhead when executing a convolution.
- We report Pareto-optimal CnM PU configurations of control and data resources across applications. Thus, we show optimal utilization of PU resources when the ratio of instruction to data capacity is set between 2 and 4.
- We analyze the integration of CnM PUs into different DRAM standards, showing the interplay among clock frequency, parallelism, and computing metrics. We find that the use of LPDDR4 can degrade the energy efficiency of CnM up to 23% with respect to HBM2, and we show that GDDR5 can achieve 77% of the performance of Hynix-AiM [14], an application-specific design.

The introduced framework is available at GitHub.

## II. Related Work

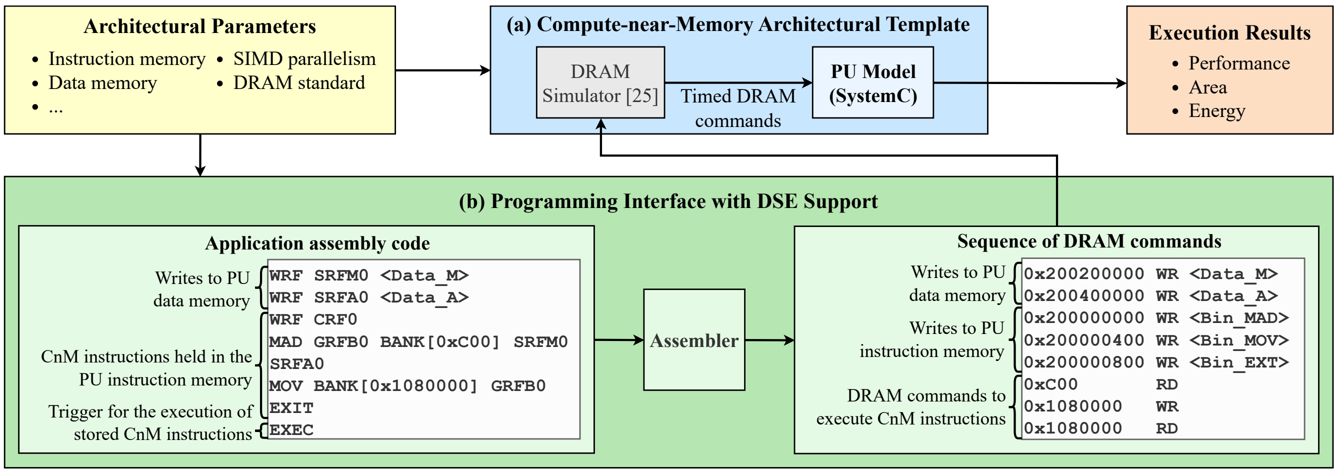
Compute-near-DRAM works have proposed to interface the processing elements at different levels of the memory structure [10], as illustrated in Fig. 1: (a) channel, (b) die, (c) bank or (d) subarray. Channel-level CnM (i.e., out of the DRAM die) interfaces a processing unit (PU) with multiple DRAM dies via an interposer [18], 3D integration [7] or the DIMM interconnect [19]. CnM at the die level places a PU within the die, shared between the banks [4]–[6]. Computing near the DRAM bank involves interfacing a PU to the banks IO [12], [14]–[17]. Finally, subarray-level CnM adds processing logic to each of the subarrays in the DRAM bank [20], [21]. The choice of integration level rests on the trade-off between computation potential and design effort. Although computing at lower DRAM levels allows a higher degree of parallelism and reduces energy consumption, it also increases the design effort due to resource constraints [9], [10]. Among these alternatives, bank-level CnM stands out as a trade-off between performance and cost [10]. This approach allows high-bandwidth and low-energy access to the stored data without modifying the internal bank structure. However, the design of CnM architectures at the bank level needs to address the stringent area limitations in the DRAM context, where resource overhead is expensive [11].

State-of-the-art industrial bank-level CnM solutions show a variety of objectives in their architectural design. UPMEM [12], [13] targets flexibility by implementing complex multi-threaded PUs with a rich ISA, as well as large local instruction and data memories. Computation is handled via a memory-mapped control interface in each DDR4 die. On the contrary, Hynix-AiM [14] and McDRAMv2 [15] focus only on deep learning workloads. Hynix-AiM accelerates matrix-vector multiplication employing dot product PUs attached to GDDR6 banks. It also implements a data memory and a look-up table to compute activations at each DRAM die, shared among banks. McDRAMv2 integrates systolic arrays in its processing units to accelerate matrix-matrix multiplication within LPDDR4 memories. These PUs also include large data memories and perform computation of common ML layers and activation functions. Hynix-AiM and McDRAMv2 completely avoid the use of instruction memories by handling execution through a modified interface with DRAM. In between these works, FIMDRAM [16] and LPDDR-PIM [17] strike a trade-off between flexibility and kernel-specific performance. Their PUs implement a simple SIMD pipeline and include small instruction and data memories.

However, the works above [12], [14]–[17] lack an analysis of the underlying design space. Filling this gap, we present a simulation framework that allows designers to perform architectural analysis of bank-level CnM solutions that execute domain-specific workloads. Consequently, it enables hardware-software co-design from the CnM system perspective. Unlike existing simulators [12], [14], [15], [17], [22] it supports the easy configuration of the CnM architectural parameters, including the datapath design and the DRAM banks. Furthermore, we ensure JEDEC-compliance as required in bank-level CnM, which published CIM exploratory frameworks [23], [24] have not addressed. Through two studies on the storage resources of CnM PUs and the choice of the DRAM standard, we demonstrate the flexibility and potential of the framework to guide design choices of CnM architectures. Our first study focuses on the size of data and control resources, not explored in previous work [12]–[17]. We showcase the trade-offs between area, performance, and energy consumption, and provide Pareto-optimal points for the first time for different target metrics when executing relevant ML and data processing kernels. Next, while previous designs focus on single DRAM configurations [12]–[15], [17], we explore how the implementation and performance of CnM processing units are affected by the choice of DRAM standard. This study depicts that CnM parallelism and working frequency are governed by DRAM specifications. For example, we illustrate that GDDR5 memories can enable CnM performances close to application-specific CnM designs [14], and that CnM with LPDDR4 memories displays higher energy overheads than HBM2 due to a lower performance that makes static power consumption prominent.

## III. Compute-near-Memory DSE Framework

Our framework, depicted in Fig. 2, is composed of **(a)** a CnM architectural template that provides a configurable model


Architectural Parameters
• Instruction memory • SIMD parallelism
• Data memory • DRAM standard
• ...
(a) Compute-near-Memory Architectural Template
DRAM Simulator [25]
Timed DRAM commands
PU Model (SystemC)
Execution Results
• Performance
• Area
• Energy
(b) Programming Interface with DSE Support
Application assembly code
Writes to PU data memory
CnM instructions held in the PU instruction memory
Trigger for the execution of stored CnM instructions
Assembler
Sequence of DRAM commands
Writes to PU data memory
Writes to PU instruction memory
DRAM commands to execute CnM instructions

```
WRF SRFM0 <Data_M>
WRF SRFA0 <Data_A>
WRF CRF0
MAD GRFB0 BANK[0xC00] SRFM0
SRFA0
MOV BANK[0x1080000] GRFB0
EXIT
EXEC
```

```
0x200200000 WR <Data_M>
0x200400000 WR <Data_A>
0x200000000 WR <Bin_MAD>
0x200000400 WR <Bin_MOV>
0x200000800 WR <Bin_EXT>
0xC00       RD
0x1080000   WR
0x1080000   RD
```

Fig. 2. Overview of the proposed Compute-near-Memory framework, allowing simulation of ML and data processing compute kernels for performance, energy and area estimates. The framework comprises the (a) Architectural Template modeling the behavior of the DRAM and PU according to design parameters, executing an application interpreted by the (b) Programming Interface.

of a processing unit (PU) attached to the DRAM banks, and **(b)** a programming interface with DSE support that interprets an application written in assembler to be executed on an instance of the CnM template. The CnM architectural template itself comprises a tunable SystemC model of a PU, which allows to explore design trade-offs and to synthesize specific instances; and a DRAM simulator [25], which provides the timing of the sequence of DRAM commands in Fig. 2.(b), conforming to compatible JEDEC standards, that trigger CnM execution in the PU model.

## A. *Compute-near-Memory Architectural Template*

As depicted in Fig. 3, the architectural template defines a processing unit interfaced with two DRAM banks (A and B). The PU, described in Section III-B, implements a single-instruction-multiple-data (SIMD) pipeline that supports addition, multiplication, Multiply-Add (MAD), and Multiply-Accumulate (MAC) operations, which are widely present in data-intensive applications. It also implements simple control and data movement instructions, as defined in Section III-C. By instantiating one PU per every two banks and exploiting concurrent access to all banks in a channel, this architecture enables massively parallel execution. The template supports integration with different DRAM standards, as described in Section III-D. The architectural template is inspired by industry-proven FIMDRAM [16], which facilitates a domain-specific starting point and ensures compliance with JEDEC standards [26]–[29]. The template generalizes the FIMDRAM design through parameter tuning to enable the extraction of prevailing trends, and can emulate it as a specific instance.

## B. *Processing Unit Architecture*

To support the functionality of the architectural template, the processing unit is composed of three main elements, shown in Fig. 3: (1) register files holding scalar and vector data, as well as CnM instructions, (2) a SIMD arithmetic unit (AU) capable of performing multiplication and addition, and (3) a control unit (CU) managing the execution of the instructions, and the interface with the host CPU and the memory banks.

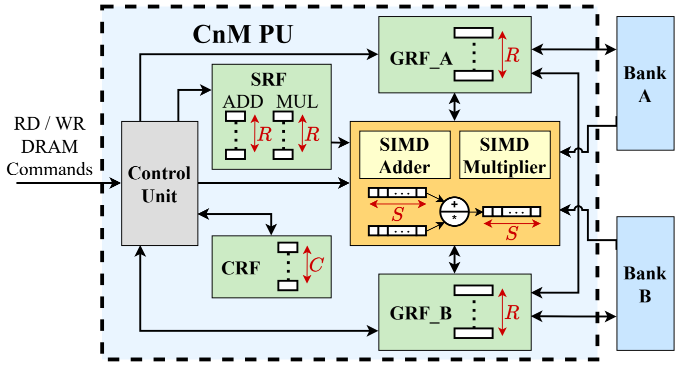


Fig. 3. Architecture of the Compute-near-Memory (CnM) processing unit (PU) and its interface to the banks. The design-time tunable parameters are highlighted in red.

Four different register files are located within the PU. First, the control register file (CRF) holds up to $C$ 32-bit instruction words to be interpreted by the CU, acting as a local instruction memory. The scalar register file (SRF) can store $R$ scalar variables for multiplication and $R$ for addition, which are replicated in the AU for every SIMD lane. Finally, two vector register files (general register files, GRFs) are present, each with capacity for $R$ vectors of $S$ words. $C$ and $R$ hence indicate the amount of resources devoted to instructions and data. We explore their interplay in Section V. As depicted in Fig. 3, GRF_A and GRF_B are interfaced to the banks A and B, respectively, to allow for direct data movement. The number of SIMD lanes $S$ and the data type should be chosen to correspond with the width of the bank IO ($S \times word\_bits = IO\_bits$). For instance, if implementing a 16-bit data type, $S$ needs to be set to 16 in accordance with the 256-bit HBM2 bank IO [26]. As the choice of DRAM standard alters the interface between the PU and the bank, Section VI analyzes the effects of bandwidth changes on computing behavior. In the PU, data movement is supported among data register files. An optional SIMD ReLU operation is also enabled when moving data to a GRF.

The arithmetic unit comprises $S$ multipliers and $S$ adders that perform in lock-step for SIMD execution. Inputs can be obtained from the GRFs, the SRF, or either of the interfaced

TABLE I
INSTRUCTIONS SUPPORTED BY THE CNM ARCHITECTURAL TEMPLATE.

| Instruction | Description |
|---|---|
| **NOP** CLKS | Multi-cycle pipeline stalling. |
| **JUMP** ADDR ITER | Repeated jump back for looping. |
| **EXIT** | End of the CnM execution. |
| **MOV** DST SRC RELU | Data movement among register files and between GRFs and banks, with optional ReLU. |
| **ADD** DST SRC0 SRC1 | Addition. |
| **MUL** DST SRC0 SRC1 | Multiplication. |
| **MAD** DST SRC0 SRC1 SRC2 | Multiply-and-add. |
| **MAC** DST SRC0 SRC1 | Multiply-and-accumulate. |

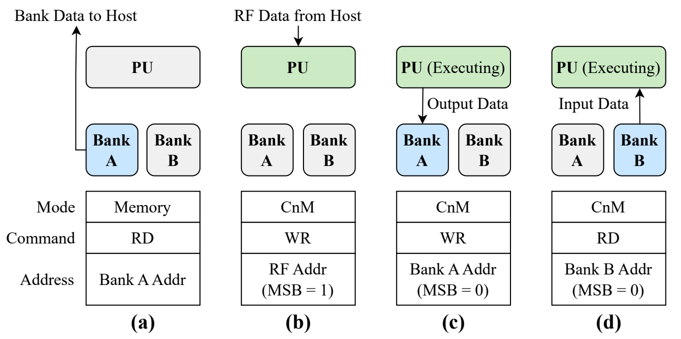


Fig. 4. State of the interface between the DRAM banks and the PU when **(a)** reading data from DRAM in memory mode, **(b)** writing to the PU registers, **(c)** executing an instruction that writes back to the bank A, and **(d)** executing an instruction that reads from bank B.

banks. The result of the operation is written back to one of the GRFs. Additionally, to allow for MAD and MAC, the output of the multipliers can be supplied into the adders.

Finally, the control unit is in charge of the execution flow. It comprises the interface with the DRAM commands that govern execution (described in Section III-D), and the logic to retrieve and decode the instructions in the CRF.

### C. Processing Unit ISA

When the execution of the processing unit is directed by the DRAM commands, a 5-stage pipeline is triggered: (1) Decode of an instruction, (2) Load data from bank, (3) Multiplication, (4) Addition, and (5) Writeback to GRF or bank. After it starts, the pipeline advances using the memory clock without requiring further DRAM commands. Any of the stages after Decode can be skipped if they are not needed for the executed instruction, e.g., instructions can skip the Load stage if they do not involve reading from a DRAM bank. As in FIMDRAM [16], the PU pipeline implements three types of instructions (described in Table I) which support the execution of linear algebra kernels present in a broad set of applications, including ML, as shown in Section IV-A. Flow-control instructions (NOP, JUMP and EXIT) guide general CnM execution. Next, data movement in the PU is handled by MOV instructions. Lastly, arithmetic instructions enable SIMD addition, multiplication, MAD, and MAC.

### D. Interface between Host and DRAM Banks

To support Compute-near-Memory, the host can alternate between two DRAM operation modes: memory mode and CnM mode. Memory-mapped registers are employed to manage mode changes. In memory mode, the DRAM acts as a normal memory and the processing units are inactive, as shown in Fig. 4(a). During CnM mode, instead, the PUs are active, and concurrent access to all banks is enabled, i.e. a single DRAM command handles the behavior of all the banks in the memory channel. This mechanism allows to govern the execution of the PUs using standard DRAM commands, avoiding modifications in the memory controller or in the interface between host CPU and memory. Hence, computation near-memory is managed by issuing read (RD) and write (WR) commands to the correct addresses, which simultaneously arrive at the banks and the PUs.

The DRAM command and the address trigger both writes to the memory-mapped PU registers (Fig. 4(b)) and execution of instructions (Fig. 4(c,d)). Specifically, the DRAM address is extended by one bit so that the new most significant bit (MSB) determines which of the two actions is performed. To guarantee synchronization between PU execution and access to the correct data in DRAM [10], [16], we assume that reordering of commands and squashing of reads during CnM mode are avoided at the memory controller.

Supporting concurrent access to all banks involves modest modifications to the memory controller. Since memory operations cannot be pipelined across bank groups, consecutive commands need to comply with the longer timings for same-bank access. To simulate this behavior and support a wide range of protocols, we extended an open-source DRAM simulator, Ramulator [25], to (1) model the channel-wide scope of memory accesses, (2) reflect the scheduling modifications due to simultaneous bank access, and (3) monitor the state of the DRAM rows, i.e., whether they are open or closed.

### E. Programming Interface

The execution of an application on the Compute-near-Memory architectural template requires a corresponding sequence of DRAM commands governing data movement and PU operation. To generate such a sequence while abstracting from the ISA implementation and memory-mapping aspects, the programming interface presented in Fig. 2(b) is employed. Since its flexibility matches that of the template, the programming interface allows to sweep the available architectural parameters for each executed application, supporting a fast exploration of the design space.

The application is implemented using a custom assembly language with a reduced number of instructions. These instructions comprise the ISA defined in Table I, commands to write to the different register files in the PU, and an EXEC instruction to generate the DRAM sequence which will trigger the CnM execution. Taking the application code as input, the assembler generates the corresponding sequence of DRAM commands, as depicted in Fig. 2(b). In addition to writing the local data memories, the sequence of CnM instructions is written into the CRF starting from the index specified in the code. Afterwards, to generate the DRAM commands to trigger execution, the instruction memory is considered from the initial index until the first EXIT command. If processing a JUMP instruction, the assembler generates the corresponding commands to decode the instructions in every iteration.

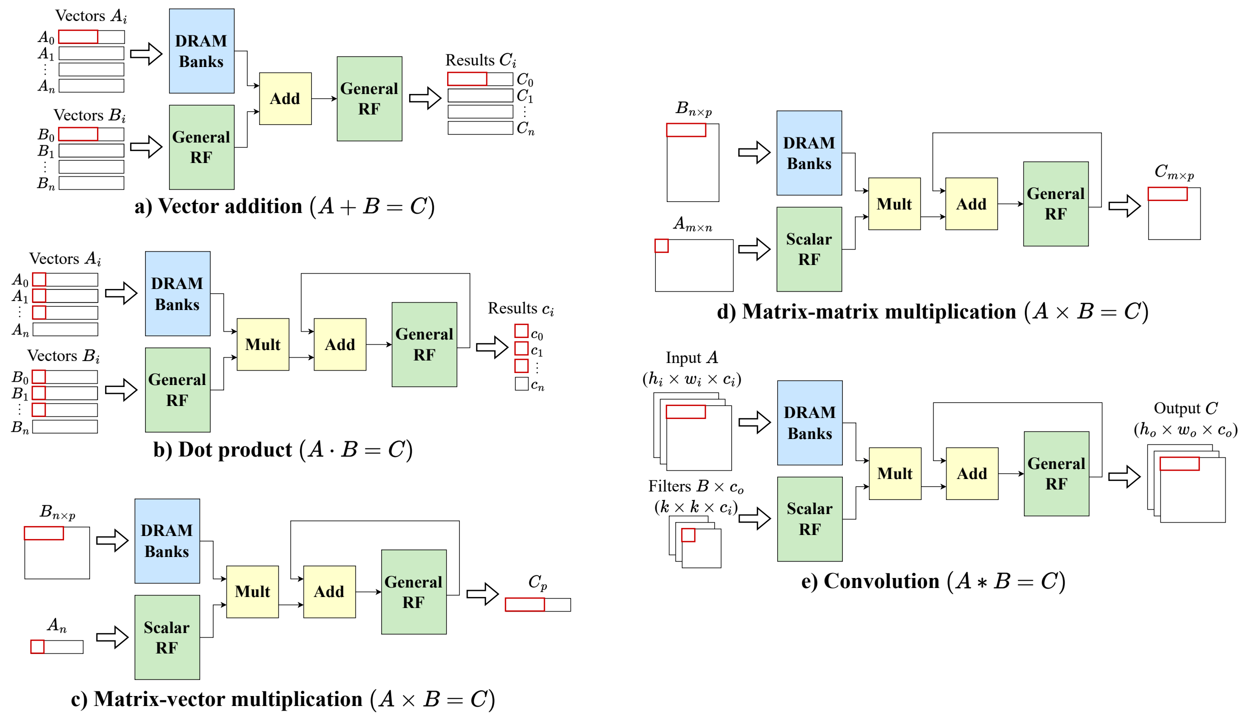


Fig. 5. Mapping of the different kernels to the CnM architecture. The red rectangles show the tilings used for computation, i.e. the elements stored in one vector or scalar register at a time.

### F. Compute-near-Memory Execution

To execute an application on the Compute-near-Memory processing units, first the programming interface is employed to obtain the input sequence of DRAM commands. The memory controller, modeled by Ramulator [25] with the extensions described in Section III-D, receives this sequence and schedules the commands according to the selected DRAM standard and the concurrent access to all banks. Consequently, it inserts the additional commands required to abide by DRAM protocols —such as activation, precharge, and refresh—, and specifies the cycle when each command is issued. The resulting timed sequence arrives at the memory banks and the PUs, modeled by the instance of the architectural template in Fig. 2(a) with the chosen architectural parameters. There, the commands are interpreted at the corresponding cycle to perform writes to the PU registers and to trigger the execution of the instructions that implement the application.

Thanks to the matching configurability of the template and the interface, an application can be executed on different architectural instances without any modification of the code. As a result, multiple instances of the execution process can be simulated, sweeping architectural parameters to perform a rapid DSE. Such flexibility enables the assessment of data representation, level of integration and usage of resources in the CnM context. In the following sections, we showcase the latter option, key in CnM system design, by analyzing the configuration of PU instruction and data capacity, along with the impact of interfacing to different DRAM standards.

## IV. Experiments

### A. Kernels Mapped to the CnM Architecture

The programming interface described in Section III-E allows the parameterized implementation of different kernels using our bank-level CnM architectural template. Here, we provide the mapping of five kernels, shown in Fig. 5: vector addition, dot product, matrix-vector multiplication, matrix multiplication, and convolution. These linear algebra and data processing operations are widely present in machine learning and scientific computing workloads where memory communication bottlenecks are frequent, e.g., in transformer models [17]. They also allow to study the behavior of both 1D and 2D kernels, which exhibit different requirements for computing and communication.

The **vector addition** kernel sums $V$ pairs of $n$-dimensional vectors. Every vector is stored in memory in row-major order, so that every DRAM column contains $S$ consecutive dimensions of a vector. As shown in Fig. 5(a), to execute the kernel, the processing unit moves the first element of each vector pair to the general registers, and adds them together with the corresponding second element obtained from the banks. The results are stored back in memory.

The **dot product** kernel performs $V$ dot product operations between two groups of $V$ $n$-dimensional vectors. Each vector is transposed and stored in memory in column-major order. During execution, the PU moves the first vector group to the GRF. Then, the elements of this group are multiplied by the corresponding ones from the second group obtained from DRAM, and the result is accumulated in one of the general registers to progressively obtain the dot product results, as depicted in Fig. 5(b).

The **matrix-vector multiplication** kernel multiplies a vector $A_n$ by a matrix $B_{n\times p}$ to obtain the vector $C_p$. The scalar RF holds the elements of $A$, while $B$ is stored in DRAM. To execute the kernel (Fig. 5(c)), each element of $A$ performs a SIMD multiplication with the corresponding elements of several columns in $B$, accumulating the results in the GRF.

TABLE II
CONFIGURATION OF THE DRAM STANDARDS FOR THE CNM ANALYSIS.

| | HBM2 [26] | DDR4 [27] | GDDR5 [28] | LPDDR4 [29] |
|---|---|---|---|---|
| Data rate (Gbps) | 2.4 | 3.2 | 4 | 3.2 |
| Array width | — | 8x | 16x | 16x |
| Internal clock (MHz) | 300 | 400 | 1000 | 200 |
| Number of banks / PUs | 16 / 8 | 16 / 8 | 16 / 8 | 8 / 4 |
| Bank IO interface (bits) | 256 | 64 | 256 | 256 |
| PU SIMD lanes (S) | 16 | 4 | 16 | 16 |
| Peak PU throughput (Gbps) | 76.8 | 25.6 | 256 | 51.2 |

The **matrix multiplication** kernel multiplies two matrices $A_{m\times n}$ and $B_{n\times p}$, obtaining the matrix $C_{m\times p}$ as a result. The elements of $A$ are loaded into scalar registers, while the rows of matrix $B$ are aligned and sequentially stored in DRAM. To obtain the rows $C$, the elements of $A$ multiply and accumulate the columns of $B$ in parallel, storing the results in the general register file (Fig. 5(d)).

The **convolution** kernel convolves a series of $c_o$ filters ($k \times k \times c_i$) with the input tensor ($h_i \times w_i \times c_i$), resulting in the output tensor of dimensions $h_o \times w_o \times c_o$. The weights and biases of each filter are loaded into the scalar register file, as depicted in Fig. 5(e). The elements in the SRF operate channel by channel with all the relevant input tensor elements, which are unrolled in the row orientation when stored in DRAM. The different output channels are obtained through repeated MAC operations with the corresponding filter coefficients, storing the results in the GRFs (Fig. 5(e)).

### B. Experimental Setup

We employ our Compute-near-Memory framework to analyze the performance of different PU configurations, modeled as instances of the architectural template. The DRAM simulator provides the scheduling of the DRAM commands, while the SystemC model simulates functionality. Across instances, we used the half-precision floating point format (16 bits), a common choice for efficient HPC and ML implementations [16]. In addition to the functional simulation performed with the framework, we employ Mentor Catapult to perform high-level synthesis of the SystemC-defined designs, and next we utilize Cadence Genus and Joules to obtain post-synthesis area and energy results using TSMC 28nm HPC logic technology.

We realize two explorations analyzing different architectural design dimensions. In Section V, we obtain different PU design points by varying the amount of resources devoted to controlling execution ($C$ control registers) and storing the kernels dataset ($R$ scalar and general registers). We analyze $C = \{16, 32, 64, 128\}$ registers per CRF and $R = \{4, 8, 16, 32\}$ registers per SRF and GRF. The HBM2 DRAM standard [26] with a 2.4 Gbps interface is considered for this exploration. To adapt to its 256-bit IO bank interface, the number of SIMD lanes in the datapath ($S$) is set to 16. The PU designs are synthesized targeting 300 MHz, matching the frequency of the HBM2 internal clock.

Next, we analyze in Section VI the performance of the CnM PUs when integrated into a channel of different popular DRAM standards. The HBM2 interface is set as the comparison baseline, resembling the FIMDRAM configuration [16]. DDR4 [27] is studied as a standard involving a low bandwidth interconnect. Exemplifying an alternative high bandwidth DRAM standard, GDDR5 [28] is also considered in the analysis. Finally, we evaluate LPDDR4 [29], a low power standard. The parameters employed in this second study are shown in Table II. To achieve a fair comparison frame, the exploration considers 4 Gb DRAM channels across standards, and the PU instances employ $C = 32$, $R = 8$ as sizing parameters. The CnM PUs are synthesized targeting the frequency of the internal clock in the standard, and their number of SIMD lanes $S$ is set to adapt to the bank IO interface.

TABLE III
BENCHMARK PARAMETERS EMPLOYED IN THE EXPLORATION OF PU DESIGN POINTS (SINGLE PU) AND OF THE INTEGRATION OF PUS IN CHANNELS OF DIFFERENT DRAM STANDARDS (CHANNEL).

| Kernel | Single PU | Channel |
|---|---|---|
| Vector addition | $V = 128, n = 128$ | $V = 256, n = 256$ |
| Dot product | | |
| Matrix-vector multiplication | $n = p = 180$ | $n = p = 1024$ |
| Matrix multiplication | $m = n = p = 60$ | $m = n = p = 128$ |
| Convolution | Input $= 11 \times 11 \times 34$<br>Output $= 9 \times 9 \times 16$<br>Filters $= 3 \times 3 \times 34$ | Input $= 24 \times 24 \times 32$<br>Output $= 20 \times 20 \times 32$<br>Filters $= 5 \times 5 \times 32$ |

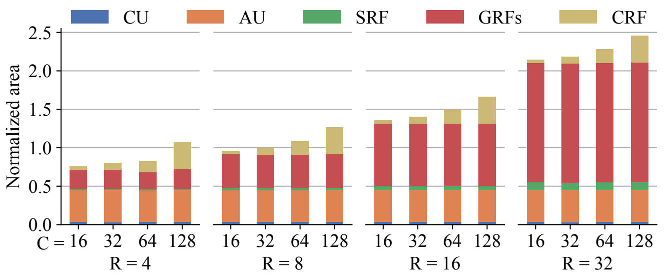


Fig. 6. Synthesized area of the components across PU configurations, normalized by the baseline design [16].

To match the design points with different application domains, we use the kernels previously described as reference points, sized as shown in Table III. To be representative of general trends for arbitrarily-sized kernels, the selected dimensions imply data mappings significantly larger than what the PUs can hold for a single kernel iteration.

## V. EXPLORATION OF BANK-LEVEL CNM PU DESIGNS

### A. Area Results

Before the studies of performance and energy consumption, we focus on the area occupation of the analyzed PU configurations to examine overhead at the confined DRAM bank periphery. This exploration allows to assess the cost of increasing computing performance via enlarging register files to improve locality. Fig. 6 shows the normalized area results after synthesis of the modeled Compute-near-Memory processing units, comprising the control unit (CU), the arithmetic unit (AU), and the register files.

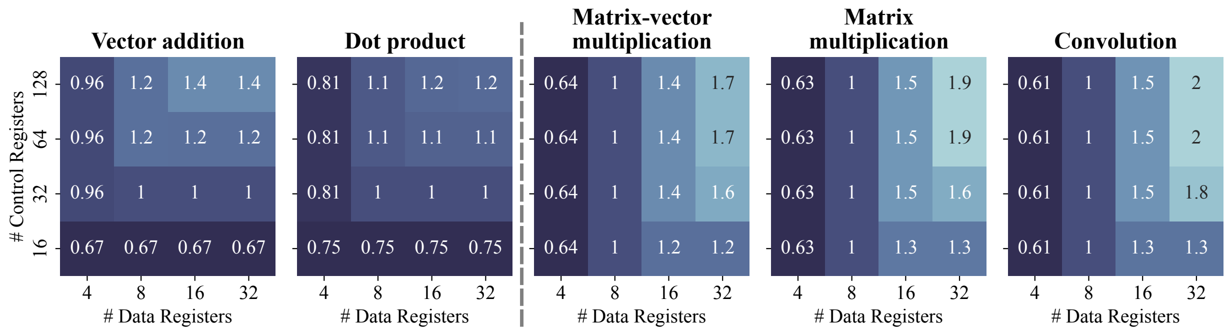


Fig. 7. Performance results (FLOPS) when executing the considered kernels, normalized with respect to the FIMDRAM [16] configuration ($C = 32, R = 8$). Vector addition and dot product (1D kernels, left) display a mainly C-limited behavior (no performance increase when adding data registers), while matrix-vector multiplication, matrix multiplication and convolution (2D kernels, right) are primarily R-limited (adding control registers fails to speed up execution).

Thanks to the simplicity of the supported instruction set, the CU has a low impact on the area of the PU. Its occupation remains constant across configurations, barely affected by the size variation of register files. The area of the AU is also stable among the studied designs, occupying a considerable fraction of the PU. However, the overhead of the design is mainly dictated by the storage elements. Particularly, GRFs rapidly dominate when increasing the number of registers to achieve better locality. Since 256-bit vector registers are employed in this round of experiments ($S = 16$), the PUs with the largest $R$ values need to accommodate up to 16 kbit of data registers. SRF occupation also presents a linear growth, but with a lower impact on area. Similarly, expanding the instruction capacity to allow the execution of more operations per iteration, reducing loop overhead, makes the CRF area significant when comprising more than 64 instructions ($> 2$ kbit).

***Key Takeaway 1*: Register files dominate the area of the PU, followed by arithmetic logic.**

Consequently, the correct sizing of the data and instruction register files is the key to obtaining good performance and energy consumption while optimizing the area of the PU, as explored in the next sections.

## *B. Performance Results*

Run-time performance of the benchmark kernels is limited by the amount of computation that can be mapped to the PU at once, executed as a loop. Larger computation tiles present a lower loop overhead and an increased data locality. In turn, the size of such tile is limited by two factors: the number of control registers $C$ and the size of the data register files $R$. The first factor defines the total instruction capacity. Instead, the amount of data registers establishes how many variables can be used in an iteration before needing an update. For a specific kernel we define $C$, $R$ configurations as $C$-limited if the number of control registers is more restrictive than $R$, or $R$-limited otherwise.

Illustrating these trends, Fig. 7 shows the performance results when executing the analyzed kernels in different configurations of CnM processing units, normalized with respect to the FIMDRAM configuration [16]. The plots demonstrate $C$-limited execution when the performance values remain unchanged when moving along the X-axis. Here, for a certain number of control registers, increasing the data capacity fails

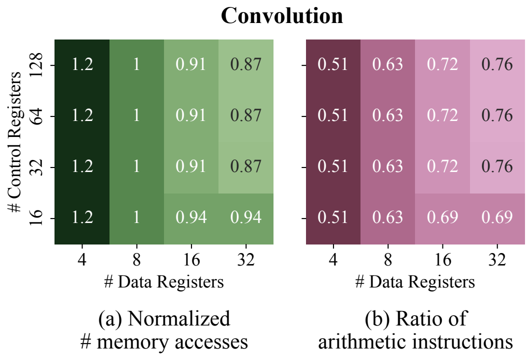


Fig. 8. Analysis of the instruction mix of the convolution kernel across $C$, $R$ configurations: (a) number of memory access normalized with respect to the FIMDRAM [16] configuration ($C = 32$, $R = 8$), and (b) utilization of the arithmetic unit, measured as the ratio between executed arithmetic instructions and the total number of instructions.

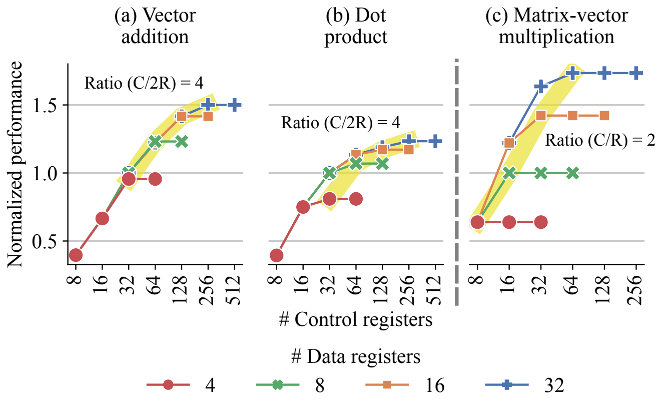


Fig. 9. Performance speedup over the baseline design [16] for representative kernels executed on PU with different $C$, $R$ configurations. The best performing ratios between instruction and data capacity are highlighted in yellow.

to achieve a performance improvement. Correspondingly, $R$-limited performance is exhibited when the values do not vary along the Y-axis.

Experiments show that the 1D kernels (vector addition and dot product) are primarily $C$-limited workloads. Due to their lack of data reuse, varying the $C$, $R$ configuration does not alter the number of memory accesses. Thus, the performance improvements when increasing instruction capacity arise from the reduction in loop overhead and the lower average latency between DRAM commands, as sequential accesses better exploit row locality. Contrarily, Fig. 7 demonstrates that the performance of matrix-vector multiplication, matrix multiplication, and convolution (2D kernels) is mostly $R$-limited. The

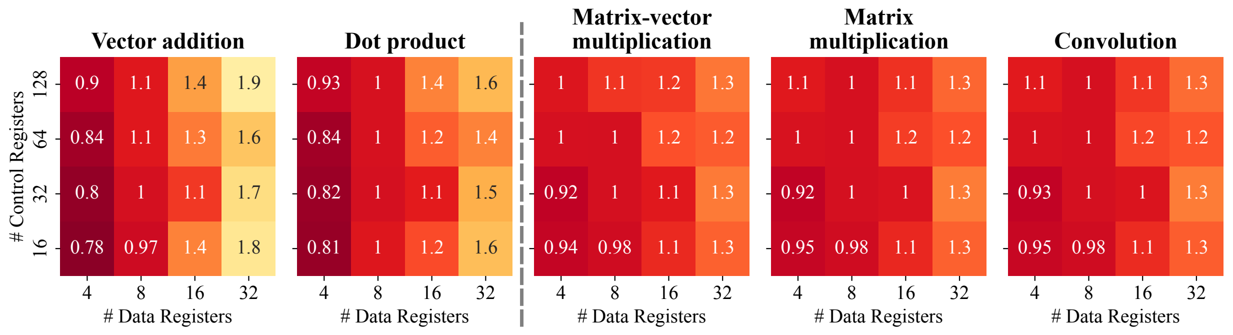


Fig. 10. Energy consumption (Joules) of a PU when executing the kernels, normalized with respect to the FIMDRAM [16] configuration ($C = 32, R = 8$). Energy use in 1D kernels (left) displays higher sensitivity to data capacity than in 2D kernels (right).

data reuse inherent to matrix operations allows to employ the contents of the data registers in several iterations before being updated. As a result, adding more data registers diminishes the number of memory accesses during execution, which in turn boosts the utilization of the arithmetic unit, as depicted in the representative convolution example in Fig. 8.

Overall, results in Fig. 7 reveal the need to balance the number of instruction and data registers in order to attain good performance at the lowest area cost, as large register files can increase overhead by more than 100%. Fig. 9 illustrates the performance change when varying the amount of control registers ($C$) for set sizes of the data register files ($R$). Across workloads, for each value of $R$ the speed-up stops growing at a certain value of $C$, at which point the instruction memory can access all the data registers in one iteration. Notably, these plateaus occur at ratios between instruction and data capacity that are consistent within the analyzed kernel. For 1D kernels, the optimal sizing ratio between instruction and data capacity is equal to **4**. This proportion allows to allocate the high number of memory access instructions per iteration required by the kernel. For example, an improvement of more than $1.6\times$ is achieved when increasing $C$ from 16 to 128 when $R = 16$, as shown in Fig. 7. Instead, 2D kernels present a lower optimal ratio of **2** between instruction and data capacity (Fig. 9(c)). The lower number stems from the presence of more arithmetic operations per memory access. In particular, multiplying the data capacity by eight achieves more than $2.6\times$ performance increase for these kernels when $C \geq 64$.

According to these results, PU designs can target different trade-offs through the sizing of register files. Maximum performance across workloads can be achieved by choosing the more limiting ratio $C/2R = 4$ on 1D kernels. However, kernels with lower optimal ratios suffer from low utilization. For example, while the configuration $C = 128$, $R = 16$ achieves the best performance in all kernels for the chosen number of data registers, more than half of the instruction capacity is unused for 2D workloads.

## C. Energy Results

The energy consumed by the PU across configurations when executing the considered kernels is shown in Fig. 10. These results illustrate a rise in energy consumption across workloads when increasing the data capacity, in dependence on the area of the processing unit and on the achieved performance. A PU covering a wide area implies both larger static power

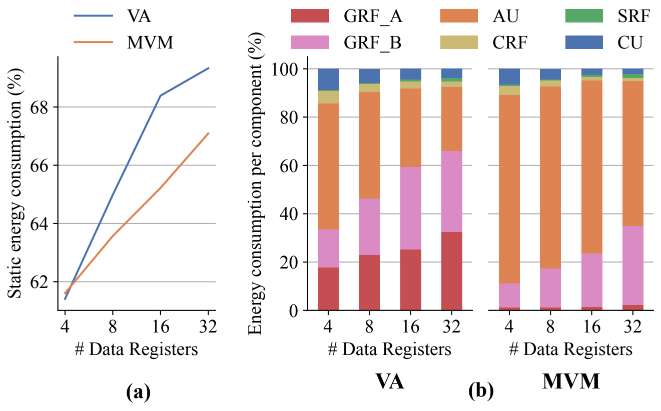


Fig. 11. Breakdown of energy results for vector addition (VA) and matrix-vector multiplication (MVM): (a) percentage of static energy with respect to the total consumption and (b) energy consumption per component.

and a higher number of components consuming switching power. However, higher-performance designs execute the kernels faster, diminishing leakage energy. At a lesser degree, results also show that, when performance is limited by $R$ or $C$, increasing the other parameter only leads to higher energy consumption due to additional leakage. Since the FLOP count for each operation remains constant across $C, R$ configurations, the heatmaps in Fig. 10 also provide energy efficiency metrics, indicating the power consumed per unit of performance (normalized W/FLOPS).

Fig. 10 depicts steeper growths in the energy costs of 1D kernels: since they are mainly $C$-limited kernels, the addition of data registers fails to significantly improve performance, and thus the static component of the power is not offset by a faster execution. Likewise, reducing $R$ has a low impact on the run-time of vector kernels, and thus higher energy savings are achieved. Instead, 2D kernels benefit from larger data RFs, as the improved performance reduces the relative energy increase. Illustrating these trends, Fig. 11(a) depicts how for vector addition the ratio of static energy grows more rapidly with the number of data registers than for matrix-vector multiplication. Besides, as 2D kernels exploit data locality at the SRF, one of the GRFs can be turned off to decrease energy consumption. Thus, the percentage of energy consumed by GRFs is larger in 1D workloads than in 2D kernels, as shown in the breakdown of per-component energy in Fig. 11(b). Consequently, energy results of 1D workloads are more sensitive to the addition of data registers. Overall, optimizing the sizing parameters can improve energy consumption by 50% for 1D kernels and by

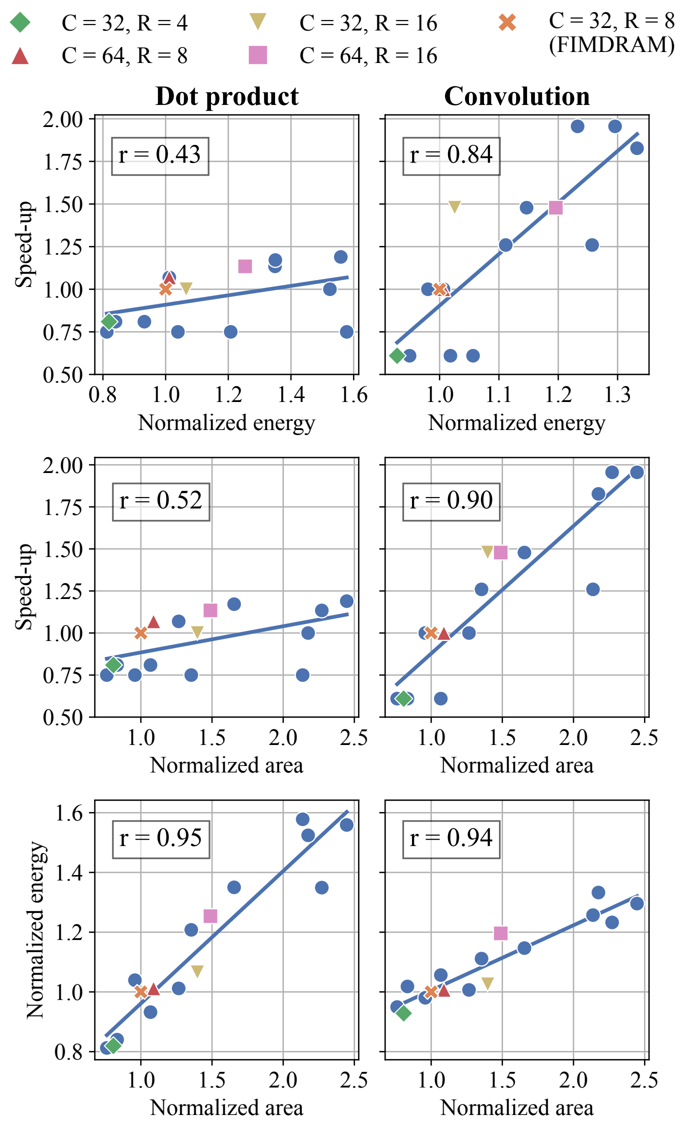


Fig. 12. Analysis of performance, energy and area trade-offs for dot product and convolution kernels. The first row shows performance vs. energy, the second row shows performance vs. area, and the third row shows energy vs. area. Highlighted PU configurations are shown in non-blue colors.

30% for 2D workloads.

***Key Takeaway 2*: Mappings that minimize the use of register files reduce the energy overhead in loop kernels.**

## *D. Performance, Power and Area Trade-offs*

Fig. 12 shows the performance, energy consumption and area trade-offs at the PU when executing two representative kernels for 1D and 2D operations, respectively dot product and convolution. We highlight several $C, R$ configurations showing sizing trends: “FIMDRAM” resembling the state-of-the-art design [16] $(C = 32, R = 8)$, a low power configuration $(C = 32, R = 4)$, two designs optimized for 1D $(C = 64, R = 8)$ and 2D operations $(C = 32, R = 16)$, and a configuration with good overall performance $(C = 64, R = 16)$.

In the first row of Fig. 12, the graphs show how improvements in speed-up come at an energy cost. However, increases in energy have more impact on 2D kernels than on 1D operations, as conveyed by the difference in slope and correlation coefficient. While energy increases are mainly driven by expansions in data capacity that mostly enhance the behavior of 2D computations, adding instruction registers to improve 1D operations has a lower energy overhead. These trends are depicted again in the graphs comparing area and speed-up, where again area increases have a bigger effect on the execution of 2D kernels. Finally, the last row in Fig. 12 showcases the linear relationship between area and energy requirements. The graph depicts how, for 1D kernels, adding more area causes a steeper growth in energy consumption. As shown in Fig. 11(a), this difference derives from the higher impact of static power, since more GRF resources are employed and run-time is not improved enough to offset the static power consumption.

***Key Takeaway 3*: The relations between performance, power and area are kernel-dependant and linear.**

When compared to the analyzed design points, the baseline inspired on FIMDRAM exhibits good performance at low energy and area costs, residing at the surroundings of the Pareto frontier. However, the processing unit can be modified to achieve lower power consumption and area occupation, or better performance. The low power configuration $(C = 32, R = 4)$ achieves 20% decrease in energy consumption and 19% lower area at a 39% performance cost with respect to FIMDRAM. A design doubling the number of instruction registers $(C = 64, R = 8)$ can improve performance of 1D operations in 23% with low energy (4%) and area (9%) overheads, and without affecting run-time or energy consumption of 2D kernels. In turn, multiplying by two the number of the data registers in a PU $(C = 32, R = 16)$ achieves up to 50% speed-up of 2D operations with a 40% area cost and a maximum energy overhead of 15%. Finally, performance can be improved across kernels by increasing both $C$ and $R$ $(C = 64, R = 16)$. While the area of the design increases in 48%, speed-ups as high as 50% are achieved at less than 33% energy overhead.

***Key Takeaway 4*: Area constraints near the bank oblige tuning of instruction and data capacity for CnM viability.**

Register files should be sized to allow for the instructions held at one time to make use of all the available data storage. Correspondingly, they should display ratios from $2\times$ to $4\times$ between instruction and variable capacity. By employing our framework, these favorable configurations can be assessed and identified.

## VI. Interfacing different DRAM Standards

### *A. Area Results*

As in the previous study, we first explore the area overhead of placing PU units near the DRAM banks. However, we now focus on the interaction between the parallelism, interface width, and clock frequency at the channels of the DRAM standards listed in Table II. We consider a single PU configuration that resembles the FIMDRAM design $(C = 32,\ R = 8)$ [16]. The area occupation of the processing units in the standards is analyzed in Fig. 13. The graph on the left shows total PU area in a channel, i.e., the area overhead per 4 Gb. With the same width of bank interface $(S = 16)$ and number of PUs, HBM2

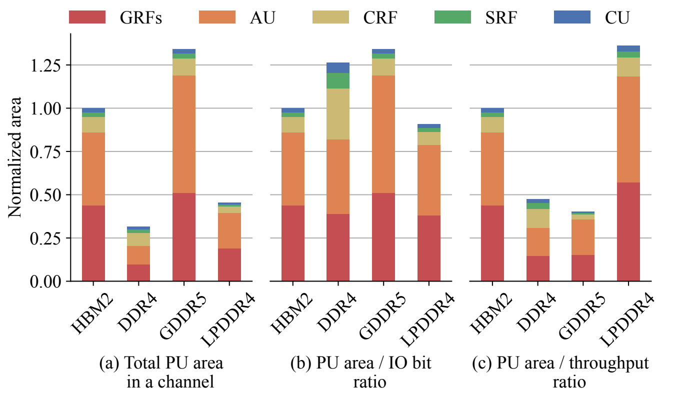


Fig. 13. Total area of the PUs in a channel across DRAM configurations (a), area per bank IO bit of the PUs in a channel (b) and area per CnM throughput across DRAM configurations (c), normalized by the FIMDRAM baseline.

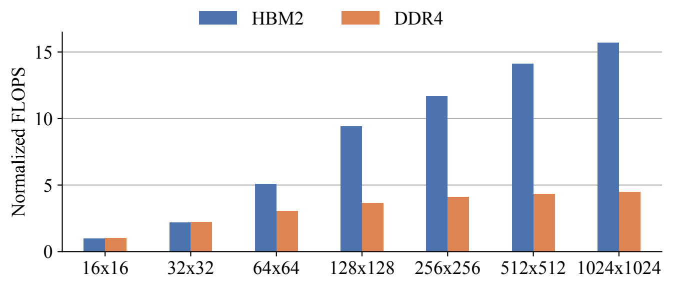


Fig. 14. Relative performance (FLOPS, the higher the better) when executing MVM with different matrix sizes, normalized with respect to the execution of 16x16 MVM employing HBM2.

and GDDR5 have equal number of interfaced PU bits (2 kbit, 128 SIMD lanes). Nevertheless, the higher clock frequency increases the area overhead in the GDDR5 channel, mainly due to the critical paths in the arithmetic unit. DDR4 has one fourth of the interfaced channel bits of HBM2 (32 lanes), resulting in the lowest area overhead despite its faster 400 MHz internal clock. Instead, for LPDDR4 the reduced overhead due to the 1 kbit (64 lanes) interfaced is further decreased by the low 200 MHz internal frequency.

The PU area per interfaced bit is shown in Fig. 13(b). Here, the effects of clock frequency are better perceived and the disparities due to different total number of PUs and SIMD lanes are concealed. Among the standards, the size of the AU and the GRFs vary according to the clock frequency. Besides, the DDR4 standard shows a larger control overhead, since the employed PUs compute using four SIMD lanes, instead of the 16 lanes used in the rest of standards.

Fig. 13(c) displays area results per peak PU throughput. Considering the values in Table II, we show that, for DDR4 and GDDR5, the area overhead is offset by the high throughput achievable with their faster clock frequencies. Likewise, LPDDR4 has low area efficiency due to its limited throughput.

***Key Takeaway 5*: The DRAM standard determines the achievable throughput between bank and CnM PU.**

Results show that, when considering a single DRAM channel, the analyzed standards offer different trade-offs. DDR4 memory achieves low area overhead and high area efficiency, while falling short in throughput due to the low total number of SIMD channels. Contrarily, high parallelism and clock frequency allows GDDR5 to attain very high throughput and area efficiency, but with a large area overhead. As the middle ground, HBM2 shows reasonable throughput and area overhead with a lower area efficiency than GDDR5, thanks to its high parallelism. LPDDR4 fails to surpass the alternatives in any metric due to its low frequency and parallelism.

However, different trade-offs are found when analyzing from the perspective of the total DRAM memory in a system. DDR4 and GDDR5 standards can increase memory and processing capacities by adding more dies to the channel. Nevertheless, in these standards supporting extra dies requires a larger total footprint and adds complexity to the interconnects with the computing elements. Since HBM2 cubes are manufactured by stacking multiple dies, this standard can support up to 16 channels with a smaller footprint and a denser interconnect through an interposer. These features allow computing near the HBM2 banks to achieve a higher throughput while keeping the area overhead reasonable. LPDDR4 dies include two channels, doubling their storage and computing capabilities. However, such increase fails to match the throughput or footprint efficiency of other DRAM alternatives.

## *B. Performance and Energy Results and Trade-offs*

All previous results referred to the large kernel dimensions in Table III to leverage CnM massive parallelism. To illustrate the results of instead employing constrained input sizes, Fig. 14 shows the interaction between dimensions of a matrix-vector multiplication kernel and the employed DRAM standard. When executing small kernels, the lower number of SIMD lanes in the DDR4 channel results in a higher fraction of active lanes than in the PUs of the HBM2 channel. For instance, MVM computation with a $32 \times 32$ matrix can be parallelized over 32 lanes, which represent the total number of SIMD lanes in DDR4, but only one fourth for the lanes in HBM2. As a result, the same number of DRAM commands is needed for executing the kernel in both standards, and the higher clock of DDR4 offsets the lower parallelism offered to match HBM2 performance. Nonetheless, when larger kernels are employed, all the SIMD lanes in the HBM2 are used, and thus the higher parallelism reduces the amount of DRAM commands needed for execution. Fig. 14 illustrates the stabilization of speed-up for large kernels, where the performance difference represents the interplay between the level of parallelism and the processing frequency. In both standards, smaller performance increases are experienced as workloads grow further due to the diminishing control overhead.

***Key Takeaway 6*: The lower bound of CnM speed-up depends on workload size. The upper bound depends on the parallelism set by the DRAM standard.**

To compare the compute-near-memory execution of the considered benchmarks using different DRAM standards, Fig. 15 displays performance, energy consumption and energy efficiency (FLOPS/W) values normalized with respect to HBM2 measurements. Speed-up values show that GDDR5 outperforms HBM2 in all kernels due to its higher clock frequency. In turn, DDR4 and LPDDR4 do not match HBM2 speed due to their low parallelism.

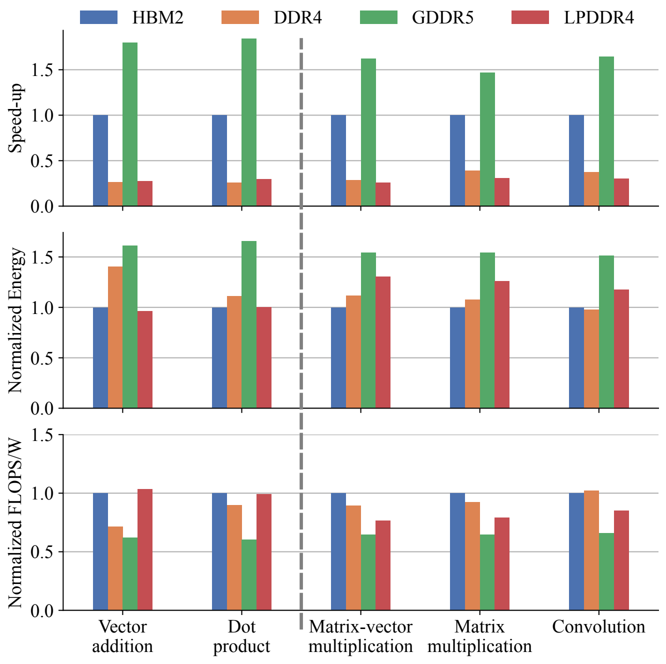


Fig. 15. Performance results when executing the studied kernels, sized as described in Table III.

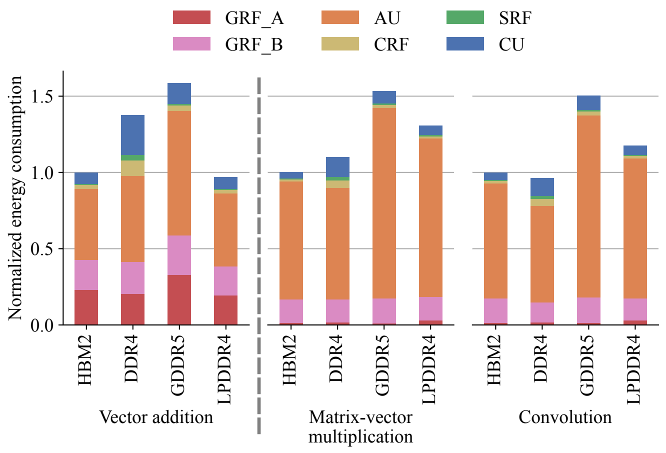


Fig. 16. Energy consumption per component when executing different kernels, normalized by the FIMDRAM baseline [16].

As for energy consumption, HBM2 and GDDR5 exhibit results proportionate to the achieved performance and the employed clock. However, DDR4 and LPDDR4 display similar or higher energy numbers than HBM2 despite their low area. Per-component energy results in Fig. 16 showcase the overhead of the control unit and CRF in DDR4 due to the higher iteration count to offset the low parallelism, particularly in vector addition where data reuse is low. In LPDDR4, the high run-time intensifies the impact of static power consumption at the arithmetic unit, specially in 2D kernels.

The third graph in Fig. 15 shows how HBM2 achieves good energy efficiency across workloads. LPDDR4 and DDR4 also obtain good results in 1D and 2D kernels respectively, where low energy overhead is observed. In contrast, CnM in the GDDR5 channel necessitates high power to maintain the obtained speed-up, hampering energy efficiency.

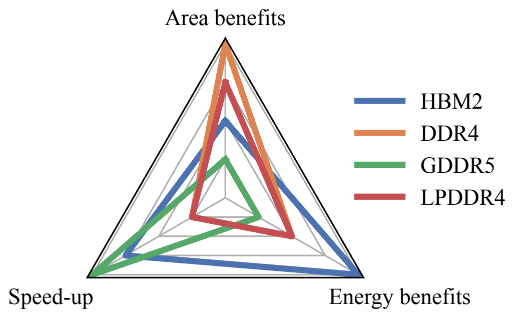


Fig. 17. Qualitative comparison of CnM metrics for different standards.

### C. DRAM Standard Trade-offs

The combined area, performance and energy results allow to optimize different design metrics through the choice of DRAM standard in CnM architectures, as qualitatively illustrated in Fig. 17. If performance is the focus, GDDR5 offers the lowest run-time when executing different kernels, though it increases area overhead and energy consumption. Instead, HBM2 trades some performance to reduce power and area overheads. It also allows to increase parallelism with the same device footprint thanks to 3D stacking, thus improving performance with respect to GDDR5 while maintaining a better energy efficiency. Finally, DDR4 and LPDDR4 offer lower area overhead alternatives, but with low performance and energy efficiency dependent on the executed kernel. In order to exploit the low frequency clocks in these standards to obtain low power designs, more area should be employed to increase parallelism.

## VII. Comparison with Recent CnM Designs

In Table IV, we compare state-of-the-art bank-level Compute-near-Memory architectures that execute matrix-vector multiplication kernels as reported in the literature [12]–[15]. To show how different configurations can be derived with our framework, we include the four designs highlighted in the PU exploration in Section V: low power ($C = 32, R = 4$), optimized for vector ($C = 64, R = 8$) and matrix operations ($C = 32, R = 16$), and good overall performance ($C = 64, R = 16$). We also cover the designs studied in the DRAM exploration in Section VI.

CnM processing units use the internal clock specified by the standard employed. In the case of McDRAMv2, a clock divider increases the operation frequency of the MAC units in the systolic array, while the rest of the PU elements use the 250 MHz clock. The size of the instruction memories determines the execution flexibility of the PU. At one end of the spectrum are the large instruction memories of UPMEM [13], which supports a complex ISA. On the other hand, Hynix-AiM [14] and McDRAMv2 [15], oriented to deep learning, avoid the use of instruction memories by allowing control of PU execution via customized DRAM commands. FIMDRAM [16] and our designs are in the middle, able to target the execution of small kernel loops in the ML and data processing domains. Similarly, the size of data memory conditions the degree of data reuse possible during PU execution. Again, UPMEM shows its aim of flexibility in the large data memories it implements, while the rest of the designs have smaller ones for the data reuse

TABLE IV
COMPARISON OF STATE-OF-THE-ART BANK-LEVEL CNM DESIGNS AND PERFORMANCE WHEN EXECUTING MATRIX-VECTOR MULTIPLICATION KERNELS.

| Design | | DRAM type | Data rate | PU clock | PU instruction memory | PU data memory | Data type | PUs per channel | SIMD lanes per PU | PU Peak Throughput | MVM 1 PU performance | MVM 1 channel performance |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| UPMEM [12], [13] | | DDR4 | 2.4 Gbps | 350 MHz | 24 kB | 64 kB | INT32 | 64[1] | 1 | 11.2 Gbps | 10.5 MOPS | 381 MOPS[1] |
| Hynix-AiM [14] | | GDDR6 | 2 Gbps | 1 GHz | — | 2 kB[2] | BF16 | 16 | 16 | 256 Gbps | 1.42 GFLOPS | 22.8 GFLOPS |
| McDRAMv2 [15] | | LPDDR4 | 3.2 Gbps | 1 GHz | — | 8.2 kB | INT8 | 4 | 128 | 64 Gbps | 74.7 GOPS[3] | 598 GOPS[3] |
| FIMDRAM [16] | | HBM2 | 2.4 Gbps | 300 MHz | 128 B | 544 B | FP16 | 8 | 16 | 76.8 Gbps | 846 MFLOPS | 10.8 GFLOPS |
| **This work** | $C = 32, R = 4$ | HBM2 | 2.4 Gbps | 300 MHz | 128 B | 272 B | FP16 | 8 | 16 | 76.8 Gbps | 677 MFLOPS | — |
| | $C = 64, R = 8$ | HBM2 | 2.4 Gbps | 300 MHz | 256 B | 544 B | FP16 | 8 | 16 | 76.8 Gbps | 846 MFLOPS | — |
| | $C = 32, R = 16$ | HBM2 | 2.4 Gbps | 300 MHz | 128 B | 1.09 kB | FP16 | 8 | 16 | 76.8 Gbps | 970 MFLOPS | — |
| | $C = 64, R = 16$ | HBM2 | 2.4 Gbps | 300 MHz | 256 B | 1.09 kB | FP16 | 8 | 16 | 76.8 Gbps | 970 MFLOPS | — |
| | DDR4 version | DDR4 | 3.2 Gbps | 400 MHz | 128 B | 544 B | FP16 | 8 | 4 | 25.6 Gbps | — | 3.07 GFLOPS |
| | GDDR5 version | GDDR5 | 4 Gbps | 1 GHz | 128 B | 544 B | FP16 | 8 | 16 | 256 Gbps | — | 17.5 GFLOPS |
| | LPDDR4 version | LPDDR4 | 3.2 Gbps | 200 MHz | 128 B | 544 B | FP16 | 4 | 16 | 51.2 Gbps | — | 2.79 GFLOPS |

[1]Considering one single rank. [2]Global buffer shared by the DRAM die. [3]Derived from reported DNN inference performance.

needed in the execution of small kernels. The PU throughput numbers demonstrate again a dependency on the targeted flexibility. UPMEM does not exploit parallelism within the PU, as it would add high complexity overhead to the already intricate pipeline. Instead, the rest of the designs leverage both channel and PU parallelism to achieve high performance.

Finally, the performance values display the result of the different architectural choices. McDRAMv2 exhibits the highest performance thanks to the compact data type used and its efficient application-specific architecture. Conversely, UPMEM presents the lowest performance as a result of its flexible PU design without data parallelism. Hynix-AiM, FIMDRAM, and our designs show mid-way performance values; however, Hynix-AiM lacks the adaptability to workloads outside the Deep Learning domain.

## VIII. CONCLUSIONS

Bank-level Compute-near-Memory architectures mitigate the communication bottleneck between computing elements and memory. When processing units (PUs) are interfaced to DRAM banks, they enable highly parallel and energy-efficient computation while reducing system-wide data transmissions. Nonetheless, their implementation entails the tuning of parameters in a multi-dimensional space. To assess the design trade-offs of this novel computing paradigm, in this paper, we have presented an architectural template and a methodology enabling the exploration of the bank-level CnM design space. Employing this template, we study the impact of design decisions on computing resources and DRAM standards. We analyze the balance between control and data resources of PUs, providing Pareto-optimal configurations for the execution of common ML and data processing kernels. Notably, we show that these design dimensions are key to steering the performance / energy / area trade-offs. In fact, resource utilization is maximized when local PU memories can store between twice and four times as many instructions as variables. We also show how high-bandwidth DRAM standards such as HBM2 and GDDR5 present a better performance at bank-level CnM than DDR4 and LPDDR4, while the latter two offer a lower area overhead.

## REFERENCES

[1] R. Wang *et al.*, "Processing Full-Scale Square Kilometre Array Data on the Summit Supercomputer," in *SC*, 2020.

[2] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH Comput. Archit. News*, 1995.

[3] N. P. Jouppi *et al.*, "In-datacenter Performance Analysis of a Tensor Processing Unit," in *IEEE ISCA*, 2017.

[4] H. Liu *et al.*, "Accelerating Personalized Recommendation with Cross-Level Near-Memory Processing," in *ISCA*, 2023.

[5] A. Sinha *et al.*, "DSIM: Distributed Sequence Matching on Near-DRAM Accelerator for Genome Assembly," *IEEE JETCAS*, 2022.

[6] J. Choi *et al.*, "ADC-PIM: Accelerating Convolution on the GPU via In-Memory Approximate Data Comparison," *IEEE JETCAS*, 2022.

[7] P. Das *et al.*, "ALAMNI: Adaptive LookAside Memory based Near-Memory Inference Engine for Eliminating Multiplications in Real-Time," *IEEE TC*, 2022.

[8] S. Bavikadi *et al.*, "A Review of In-Memory Computing Architectures for Machine Learning Applications," in *ACM GLSVLSI*, 2020.

[9] G. Singh *et al.*, "A Review of Near-memory Computing Architectures: Opportunities and Challenges," in *Euromicro DSD*, 2018.

[10] C. Sudarshan *et al.*, "A Critical Assessment of DRAM-PIM Architectures - Trends, Challenges and Solutions," in *SAMOS*, 2022.

[11] D. Kim *et al.*, "An Overview of Processing-in-Memory Circuits for Artificial Intelligence and Machine Learning," *IEEE JETCAS*, 2022.

[12] F. Devaux, "The true Processing In Memory accelerator," in *IEEE HCS*, 2019.

[13] J. Gómez-Luna *et al.*, "Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System," *IEEE Access*, 2022.

[14] Y. Kwon *et al.*, "System Architecture and Software Stack for GDDR6-AiM," in *IEEE HCS*, 2022.

[15] S. Cho *et al.*, "McDRAM v2: In-Dynamic Random Access Memory Systolic Array Accelerator to Address the Large Model Problem in Deep Neural Networks on the Edge," *IEEE Access*, 2020.

[16] S. Lee *et al.*, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology," in *ISCA*, 2021.

[17] J. H. Kim *et al.*, "Samsung PIM/PNM for Transformer based AI: Energy Efficiency on PIM/PNM Cluster," in *IEEE HCS*, 2023.

[18] I. Fernandez *et al.*, "NATSA: A Near-Data Processing Accelerator for Time Series Analysis," in *IEEE ICCD*, 2020.

[19] L. Ke *et al.*, "Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM," *IEEE Micro*, 2022.

[20] C. Sudarshan *et al.*, "A Novel DRAM-Based Process-in-Memory Architecture and its Implementation for CNNs," in *ASP-DAC*, 2021.

[21] S. Li *et al.*, "SCOPE: A stochastic computing engine for DRAM-based in-situ accelerator," in *MICRO*, 2018.

[22] X. Xie *et al.*, "MPU-Sim: A Simulator for In-DRAM Near-Bank Processing Architectures," *IEEE CAL*, 2022.

[23] X. Peng *et al.*, "DNN+NeuroSim: An End-to-End Benchmarking Framework for Compute-in-Memory Accelerators with Versatile Device Technologies," in *IEEE IEDM*, 2019.

[24] M. Zahedi *et al.*, "MNEMOSENE: Tile Architecture and Simulator for Memristor-based Computation-in-memory," *ACM JETC*, 2022.

[25] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE CAL*, 2016.

[26] JEDEC, "High Bandwidth Memory (HBM) DRAM." JESD235D, 2021.

[27] JEDEC, "DDR4 SDRAM Standard." JESD79-4D, 2021.

[28] JEDEC, "Graphics Double Data Rate (GDDR5) SGRAM Standard." JESD212C.01, 2023.

[29] JEDEC, "Low Power Double Data Rate 4 (LPDDR4)." JESD209-4D, 2021.