Project:

MNEMOSENE

(Grant Agreement number 780215)

"Computation-in-memory architecture based on resistive devices"

Funding Scheme: Research and Innovation Action

Call: ICT-31-2017 "Development of new approaches to scale functional performance of information processing and storage substantially beyond the state-of-the-art technologies with a focus on ultra-low power and high performance"

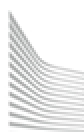
Date of the latest version of ANNEX I: 11/10/2017

D1.3 – Final report on new algorithmic solutions

Project Coordinator (PC):	Prof. Said Hamdioui Technische Universiteit Delft - Department of Quantum and Computer Engineering (TUD) Tel.: (+31) 15 27 83643 Email: S.Hamdioui@tudelft.nl
Project website address:	www.mnemosene.eu
Lead Partner for Deliverable:	TUD
Report Issue Date:	30/09/2020

Document History *(Revisions – Amendments)*	
Version and date	**Changes**
1.0 30/04/2020	Outline version
2.0 06/05/2020	Revised outline version
3.0 19/06/2020	First draft from IBM
4.0 30/06/2020	Preliminary version without IMEC content
5.0 02/10/2020	Final version

Dissemination Level		
PU	Public	**X**
PP	Restricted to other program participants (including the EC Services)	
RE	Restricted to a group specified by the consortium (including the EC Services)	
CO	Confidential, only for members of the consortium (including the EC)	



The MNEMOSENE project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant

The MNEMOSENE project aims at demonstrating a new computation-in-memory (CIM) based on resistive devices together with its required programming flow and interface. To develop the new architecture, the following scientific and technical objectives will be targeted:

- Objective 1: Develop new algorithmic solutions for targeted applications for CIM architecture.
- Objective 2: Develop and design new mapping methods integrated in a framework for efficient compilation of the new algorithms into CIM macro-level operations; each of these is mapped to a group of CIM tiles.
- Objective 3: Develop a macro-architecture based on the integration of group of CIM tiles, including the overall scheduling of the macro-level operation, data accesses, inter-tile communication, the partitioning of the crossbar, etc.
- Objective 4: Develop and demonstrate the micro-architecture level of CIM tiles and their models, including primitive logic and arithmetic operators, the mapping of such operators on the crossbar, different circuit choices and the associated design trade-offs, etc.
- Objective 5: Design a simulator (based on calibrated models of memristor devices & building blocks) and FPGA emulator for the new architecture (CIM device combined with conventional CPU) in order demonstrate its superiority. Demonstrate the concept of CIM by performing measurements on fabricated crossbar mounted on a PCB board.

A demonstrator will be produced and tested to show that the storage and processing can be integrated in the same physical location to improve energy efficiency and also to show that the proposed accelerator is able to achieve the following measurable targets (as compared with a general purpose multi-core platform) for the considered applications:

- Improve the energy-delay product by factor of 100X to 1000X
- Improve the computational efficiency (#operations / total-energy) by factor of 10X to 100X
- Improve the performance density (# operations per area) by factor of 10X to 100X

LEGAL NOTICE

Neither the European Commission nor any person acting on behalf of the Commission is responsible for the use, which might be made, of the following information.

The views expressed in this report are those of the authors and do not necessarily reflect those of the European Commission.

© MNEMOSENE Consortium 2020

Table of Contents

1. Introduction ... 4
2. Classification and comparison of memory-centric architectures ... 5
2.1 Classification of memory-centric architectures ... 5
2.2 Qualitative comparison of memory-centric architectures ... 7
3. Database query application ... 8
3.1 Review of in-memory database query algorithm ... 8
3.2 Cascaded in-memory database query ... 9
4. Matching with automata processor ... 12
4.1 Motivation ... 12
4.2 Implementation with CIM ... 13
4.3 Evaluation based on simulations ... 16
5. Image processing application ... 19
6. Deep learning inference application ... 23
6.1 Motivation: Challenges Related to Weights Polarity in Crossbar NN Accelerators 23
6.2 Hard-Constrained Quantized Training ... 24
6.2.1 Loss Definition ... 25
6.2.2 Unipolar Weight Matrices Quantized Training ... 26
6.3 Experiments and Results: Unipolar Weights vs Accuracy Trade-off ... 26
6.3.1 Fully Connected DNN: HAR ... 27
6.3.2 Deeper CNN: CIFAR10 ... 28
6.4 Energy and Area Benefits ... 28
6.4.1 Energy Estimation ... 30
6.4.2 Area Estimation ... 31
6.5 Conclusions ... 32
7. Hyperdimensional computing application ... 33
7.1 Implementation of HD computing with CIM ... 33
7.2 Associative memory search with CIM ... 35
7.3 N-gram encoding with CIM ... 39
8. Application outlook ... 42
8.1 Summary of MNEMOSENE applications ... 42
8.2 Other applications that could benefit from MNEMOSENE kernels ... 42
8.2.1 Sparse coding ... 43
8.2.2 Threshold Logic ... 43
8.2.3 Linear equation solvers ... 43
9. Bibliography ... 44

1. Introduction

A radical departure from traditional von Neumann systems, which involve separate processing and memory units, is needed in order to build efficient non-von Neumann computing systems for highly data-centric artificial intelligence related applications. The computing systems that run today's AI algorithms are based on the von Neumann architecture where large amounts of data need to be shuttled back and forth at high speeds during the execution of these computational tasks. This creates a performance bottleneck and also leads to significant area/power inefficiency. Thus, it is becoming increasingly clear that to build efficient cognitive computers, we need to transition to novel architectures where memory and processing are better collocated. Computational memory or computation-in-memory (CIM) architecture is one such approach where certain computational tasks are performed in place in the memory itself by exploiting the physical attributes of the memory devices.

In the previous deliverables of work package 1, we have described several applications that clearly and significantly benefit from CIM architecture based on non-volatile memories. These applications cover the domains of data analytics, signal processing, and machine learning. In D1.1, an outline of all the applications investigated in this work package was provided, as well as the CIM kernels they rely on and preliminary performance estimates. In D1.2, first algorithmic solutions for optimal CIM implementation of three applications were provided, that is, database query, compressed sensing, and deep learning inference.

In this final deliverable of work package 1, we provide final algorithmic solutions for the remaining applications that were not covered in D1.2, as well as additional new algorithmic solutions to the database query and deep learning inference applications. Specifically, this deliverable covers database query, matching with automata processor, guided image filtering, deep learning inference, and hyper-dimensional computing applications. Furthermore, we provide a qualitative comparison of the different CIM architectures to near-memory computing and provide an outlook of other additional applications that could benefit from the MNEMOSENE CIM kernels but were not investigated in work package 1.

2. Classification and comparison of memory-centric architectures

Memory-centric architectures based on emerging memristive devices provides super-fast performance and energy-efficiency by processing data in the storage unit. It avoids the costly data movement between processing and storage units as it is done in the traditional processor-centric architectures. Since memory-centric architectures have two fundamental modules, memory array and periphery, the computation can be performed in either of these two modules. Based on the computation location memory-centric architectures can be classified into two main categories, namely Computation-Inside-Memory (CIM) and Computation-Outside-Memory (COM). In COM architectures the computation is performed in the extra logic circuits inside the memory (SiP), and they are commonly referred as Near-memory-computing. Thus we will use the term COM-N in the remaining of this report to refer Computation-Outside-Memory array or near memory computation.

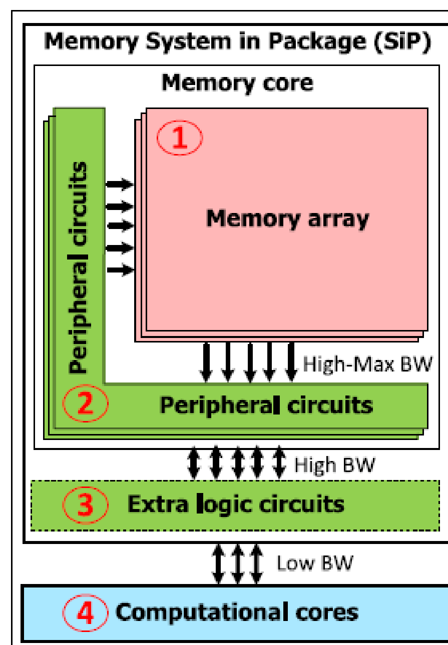


Figure 1: Computer architecture for processor and memory centric designs

2.1 Classification of memory-centric architectures

As mentioned earlier, memory-centric architectures can be divided into two classes known as CIM and COM depending on where the computation is performed. CIM architectures can be classified into CIM-Array (CIM-A) if the computation is performed within the memory array and CIM-Periphery (CIM-P) if it is performed in the periphery of the memory array. These four classes are discussed as follows:

A) **CIM-A:** In CIM-A, the computation result is produced within the memory array (noted as position 1 in Figure 1). Note that this is different from a standard write operation. CIM-A architectures always require a redesign of cells to support such logic design, as the conventional memory cell dimensions and their embedding in the bit- and

wordline structure do not allow them to be used for logic. A memory cell is namely heavily optimized in terms of processing stack and layout; hence, any changes in the array access require a completely new cell design and characterization process as the material stack of a memory array is specifically optimized for specific control voltages, current, and so forth. In addition, modifications in the periphery are sometimes needed to support the changes in the cell. Few architectures have been proposed in this class; these include Complementary Resistive Switch (CRS), Computation-in-Memory (CIM), Memristive Memory Processing Unit (MPU), Programmable Logic-in-Memory Computer (PLiM), and ReRAM-based VLIW architecture (ReVAMP). All these architectures have similar organization as they contain a memristor crossbar which serves both as a storage and computation unit.

CIM-A architectures can be subdivided into two groups: (1) basic CIM-A, where only changes inside the memory array are required, and (2) hybrid CIM-A, where, in addition to major changes in the memory array, minimal to medium changes are required in the peripheral circuit. Since the results of CIM-A architectures are produced inside the memory array, which may sometimes require readout operations to obtain the results for further calculations.

B) CIM-P: CIM-P is a memory-centric architecture in which the computation result is produced within the peripheral circuitry (noted as position 2 in Figure 1). In CIM-P computations are performed during readout operations (i.e., two or more wordlines are activated simultaneously) using special peripheral circuitry. CIM-P architectures typically contain dedicated peripheral circuits such as DACs and/or ADCs and customized sense amplifiers. Note that more radical changes in the peripheral circuit can be made in the future (e.g., changing in control voltages leads to radical changes in voltage drivers and sense amplifiers, or including a full functional processor inside memory banks). Even though the computational results are produced in the peripheral circuits for CIM-P, the memory array is a substantial component in the computations. As the peripheral circuits are modified, the currents/voltages applied to the memory array are typically different than in the conventional memory.

Similar to CIM-A, CIM-P architectures are also further divided into two categories; (1) basic CIM-P, where only changes inside the peripheral is required, which means the current levels should not be affected; and (2) hybrid CIM-P, where the majority of the changes take place in the peripheral circuit and minimal to medium changes in the memory array. In CIM-P the results are obtained directly after the operations and may sometimes need an additional step to write the results back to memory.

C) COM-N: The COM-N class consists of architectures that perform computation using additional logic units outside the memory core but inside the memory (SiP) (noted as position 3 in Figure 1). These architectures were proposed in the past and evolved through different memory technologies ranging from conventional DRAM and embedded DRAM to emerging memory technologies such as RRAM. COM-N architectures can be considered as predecessors of CIM architectures in bringing the computation towards memory-centric. However, since the computation in COM-N is still performed by logic units outside of the memory core, the data movement issue is not well addressed in COM-N architectures, which reduces their performance, energy-efficiency and band-width when compared to the CIM architectures.

2.2 Qualitative comparison of memory-centric architectures

In order to demonstrate the potential and advantages of Computing-In-Memory (CIM) architectures over their processor-centric or near memory computing (COM-N) counter parts, qualitative comparison of the architectures is performed as shown in Table 1 and Table 2.

Table 1: Comparison of memory centric architectures in terms of data movement, computation requirement, bandwidth and memory design efforts

Architecture	Data movement outside memory core	Computation requirements		Available Bandwidth	Memory design effort		
		Data alignment	Complex function		Cell & Array	Periphery	Controller
CIM-A	No	Yes	High latency	Maximum	High	Low/medium	High
CIM-P	No	Yes	High cost	High-Maximum	Low/medium	High	Medium
COM-N	Yes	NR	Low cost	High	Low	Low	Low

Table 1 compares the three architectures, namely CIM-A, CIM-P and COM-N, with respect to data movement, computation requirement, bandwidth and memory design effort. As it can be seen from Table 1 both CIM-A and CIM-P does not have data movement and hence, deliver higher bandwidth than COM-N. However, they need higher memory design effort than their COM-N counterpart.

Similarly Table 2 compares the architectures in terms of endurance, maturity and scalability. From the table one can observe that due to their emerging nature, CIM-A and CIM-P are less mature and lack software and technology support when compared to COM-N. This indicates that substantial amount of work is required for a fully-fledged utilization of CIM architecture in the future.

Table 2: Comparison of memory-centric architectures in terms of endurance, technology support, development maturity and scalability

Architecture	Endurance requirement	Maturity		Scalability
		Software support and technology	Development	
CIM-A	High	Emerging	Simulation	Low
CIM-P	Medium	Emerging	Simulation	Medium
COM-N	Medium	Commercialized	Fabricated	Medium

3. Database query application

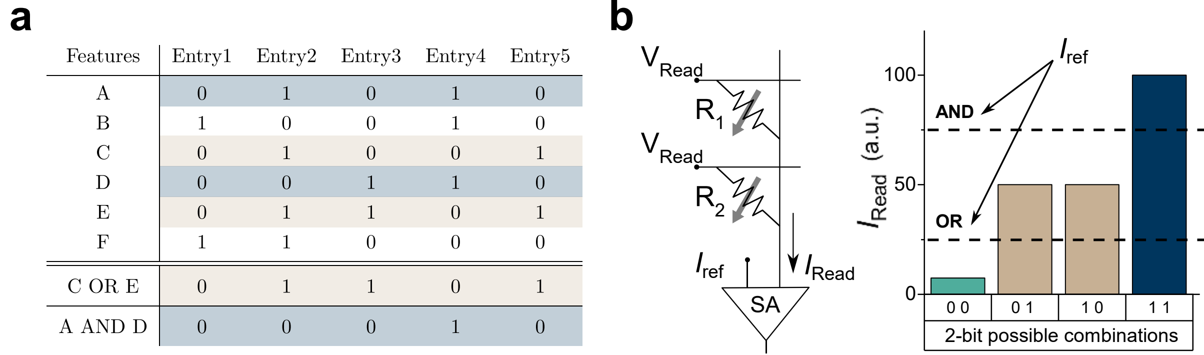
3.1 Review of in-memory database query algorithm

Query operations can be performed on databases that are structured collections of attributes or features, associated with different subject or item entries. The objective of a query is to retrieve the entries from the database that satisfy certain constraints related to the features. When databases are represented in a bitmap representation (vectors of the logical “0” and “1”), it is possible to formulate the queries as bulk bitwise operations on the feature vectors (see Figure 2a). The key idea of in-memory database query is to store the database entries in arrays of memristive devices using their conductance as the logic state variable. Subsequently, the bulk bitwise operations associated with the query operations are performed in place in the memristive arrays by employing in-memory logic.

For in-memory logic, we exploit the non-volatile binary storage capability of the memristive devices. These devices can be scaled down to nanoscale dimensions [1, 2] and their non-volatile storage capability ensures that no energy is spent to retain the stored information unlike in DRAM. For example, the 0s and 1s can be represented by the memristive devices low and high conductance states respectively. Several logical operations can be enabled through the interaction between the voltage and conductance state variables [3, 4]. For the database query problem, we resort to non-stateful logic operations where the logical operands are stored as conductance values, but the result of the logical operation is obtained as a current signal. This method of read assisted logic was first implemented in Pinatubo [5] and has inspired the concept of scouting logic [6, 7]. Therefore, the operands stay fixed in the memory array and the devices need not be programmed during the evaluation of the logical operation.

Figure 2b shows the realization of bitwise logical operations using scouting logic. Memristive devices are organised in a crossbar configuration, and by simultaneously activating multiple rows we can perform logical operations such as AND and OR. This enables us to execute the database query operations. For example, a query that requests the input entries satisfying the features “A” AND “D” could be performed by biasing simultaneously the crossbar rows corresponding to the A and D feature vectors with a specific read voltage (V_{Read}). The resulting read current (I_{Read}) along each column corresponds to the summed conductance of the memristive devices according to Kirchhoff's circuit laws. Sense amplifier (SA) per column is equipped with appropriate reference currents (I_{ref}) of appropriate choice, and the required AND logical operation can be performed by comparing I_{ref} with I_{Read}.

The logical result is thereby calculated in the same memory unit without having to move the contents to an external processing unit. Moreover, the output is a vector with equal length as the number of crossbar columns and contains the query response for all the entries, thus facilitating the execution of queries with high parallelism. Processors based on memristive devices with logic capabilities have been demonstrated in the past [8, 9], and even specifically using the scouting logic concept [10]. However, a system capable of executing in-memory a query that consists of a multitude of logical operands is lacking. In section 3.2 we introduce a novel computing system for cascaded query to execute queries of arbitrary length and complexity. It is a fully configurable digital system that can change between OR & AND according to the input query. A large-scale database query that involves a series of logical operations is presented as a case-study.



Features	Entry1	Entry2	Entry3	Entry4	Entry5
A	0	1	0	1	0
B	1	0	0	1	0
C	0	1	0	0	1
D	0	0	1	1	0
E	0	1	1	0	1
F	1	1	0	0	0
C OR E	0	1	1	0	1
A AND D	0	0	0	1	0

Figure 2 **a** An example database consists of 5 subject or item entries each of them with 6 features expressed in a binary format (typically referred to as a bitmap representation). Queries consist of performing logical operations between the feature vectors and can be executed as bitwise logical operations. **b** A schematic illustration of scouting logic employed for bitwise logical operations.

3.2 Cascaded in-memory database query

Real-world database queries consist of a multitude of sub-queries with associated logical operations rather than a single query. Solving such a query in the previously demonstrated fashion could yield an inefficient system. The main challenge is that it requires an additional memory unit for temporary storage of intermediate logical results and subsequent fetching for further processing along with the next set of logical outputs.

To address this, a configurable computing system is introduced that combines in- and near-memory computations. Note that any query can be expressed as the sum of products (SOP): $(a_1 \cdot b_1) + (a_2 \cdot b_2)$, or the product of sums (POS): $(a_1 + b_1) \cdot (a_2 + b_2)$, where sum and product operators correspond to OR and AND respectively. However, their occurrence is instructed by the query and is not necessarily alternated OR and AND operations. Hence, an arbitrary query function $F(A)$ can be expressed as a combination of POS and SOP as given by:

$$F(A) = F(S)_1 * F(S)_2 * \ldots * F(S)_p \quad (1)$$

where $F(S)_i = a_i * b_i$, $*$ is an OR or AND operator, and p depends on the query length.

The key idea of the proposed cascaded logic computing system is to perform a logical operation both in-memory and near-memory simultaneously. While an in-memory analog computation using scouting logic is executed at the memristive crossbar, a near-memory digital logic operation is carried out at the periphery of the memory array using conventional CMOS-based gates (see Figure 3a). But rather than independently computing in parallel, the system executes the decomposed query expressed in terms of Equation (1) in a cascaded manner. At a given clock cycle, the control unit selects the crossbar rows that correspond to the questioned attributes and configures the sense amplifier by selecting the appropriate reference current (switch S1), as instructed by the query operator. Subsequently, the logical results obtained at the SA output nodes are stored in a buffer. This will serve as the first input to the digital gate that can either be an OR or an AND gate depending on switch S2 that enables the corresponding flip-flop and multiplexer channel. The second input to the digital gate is the accumulated logical result of all the previously

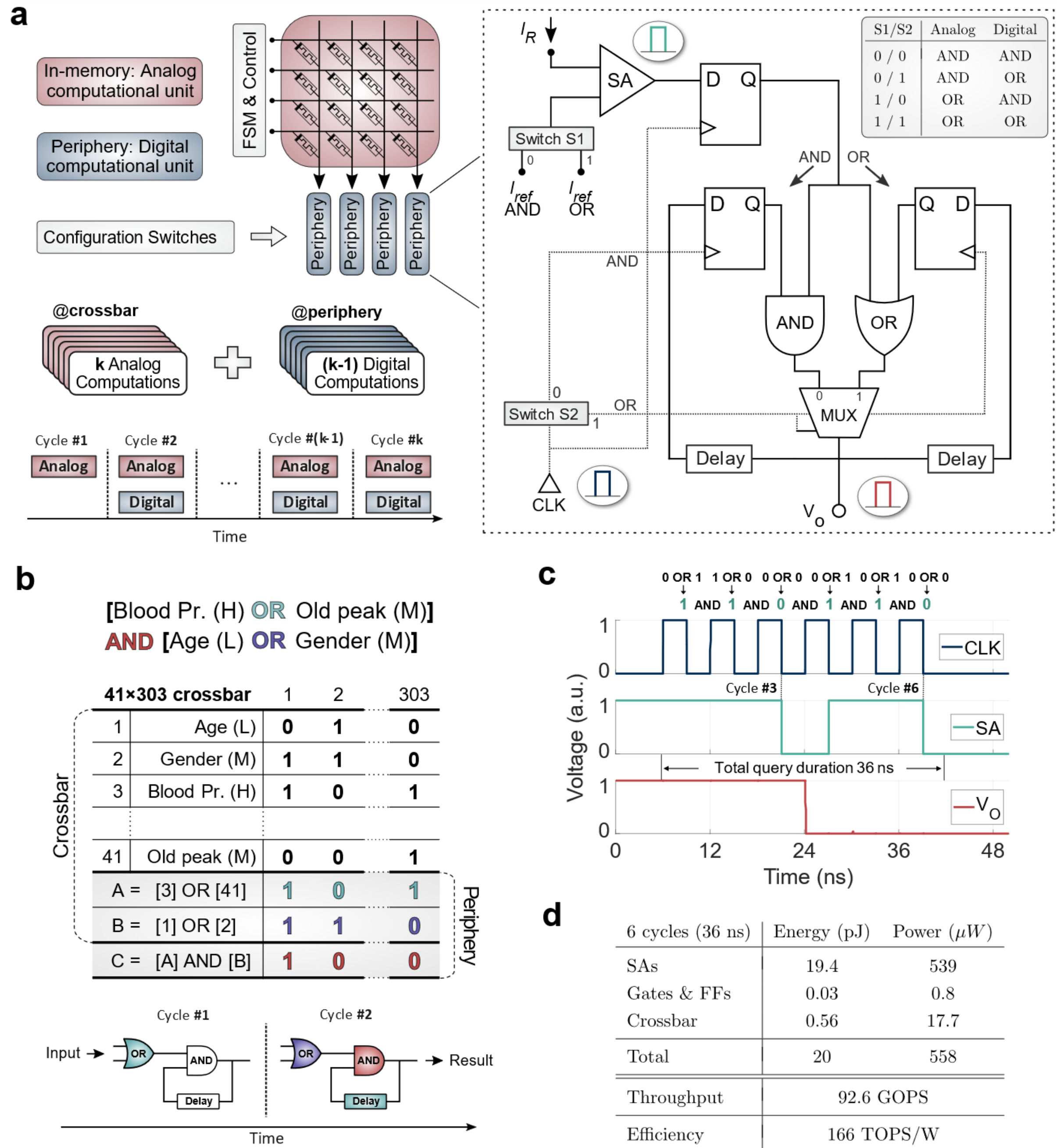
executed logical operations. The output of the digital gate will serve as the new intermittent result that gets buffered in a delay circuit. At the subsequent clock-cycle this signal will in turn be buffered and will be input to the gate, along with the new crossbar output. In other words, at every cycle, the digital output gets updated with the subsequent result of the logical operation obtained from the crossbar, until the query function gets fully executed.

To demonstrate the concept's efficacy, we encounter a practical problem with a much larger database. The Cleveland heart disease database, that is available on the UCI machine learning repository, consists of medical metrics obtained from 303 patients with heart-related health problems [11]. After binarization, the database featured 41 attributes requiring a 41×303 crossbar array. The query comprises the AND of two OR operations (see Figure 3b). At the first clock-cycle, rows "3" & "41" are biased with V_{Read} and each column current is measured by a SA that is configured to perform the OR scouting logic operation. The result of the OR operation is input to the digital AND gate. For the first cycle, the second input to the digital AND gate would be initialized to logical level 1. At the subsequent cycle, the rows "1" & "2" are activated and their partial logical result (OR) is input to the digital AND gate along with the buffered OR result from cycle #1. The final query response is the binary vector that consists of the 303 logical results, as obtained from the output nodes of the digital gates right after the end of cycle #2.

Parameters	Specifications
Array size	41x303
# of patients (entries)	303
# of attributes (features)	41
Simulator	Synopsys HSPICE
CMOS technology	TSMC 65 nm
Sense amplifier (latency)	3 ns

The system offers massive parallelism combined with no need for high-power device programming and thus is expected to execute queries with remarkable efficiency. To provide computational metrics, we set up a 41×303 crossbar in a Synopsis HSPICE circuit simulator. The device electrical characteristics were obtained from fabricated projected PCM devices (see D5.2 document). We used a current-latched sense amplifier realized on 65 nm SRAM technology [12] and our cascaded logic digital circuit was built around it. Even though the design offers delay times lower than 3 ns, the period was doubled to cover additional delays that may occur from charging the routing wires and parasitic capacitors. The simulated system was clocked at 167 MHz and configured to solve product-of-sums. We chose an example query comprising 11 consecutive OR and AND operations, out of which 6 are performed in the analog domain. A maximum of 2 operations are performed at each clock period (6 ns) which means that the total time required is 36 ns (see Figure 3c). Simulated performance metrics revealed that both the PCM-based crossbar and the digital-logic circuitry (gates, flip-flops, multiplexer) have a very low energy impact compared to the 303 SA units that dominate the time and power consumption. The total average power consumption of the system is 558 μW and the total required energy for the fully cascaded query is 20 pJ (3.3 pJ/cycle). These numbers refer explicitly to the core components, which means that the control unit and any post-processing circuits are excluded from the

calculation. The achieved throughput was 92.6 GOPS and the energy efficiency reached 166 TOPS/W. The performance metrics are summarized in Figure 3d for the total system.



6 cycles (36 ns)	Energy (pJ)	Power (μW)
SAs	19.4	539
Gates & FFs	0.03	0.8
Crossbar	0.56	17.7
Total	20	558
Throughput	92.6 GOPS	
Efficiency	166 TOPS/W	

Figure 3 **a** Schematic illustration of the cascaded system showing how the analog and digital computations are distributed to the crossbar and periphery respectively. The digital circuitry design, that locates at each column node, consists of a sense amplifier (SA), CMOS logic gates, a multiplier and the switches that configure the circuit according to the query-instructed operation. SA converts the scouting logic analog result from a crossbar column and feeds it to the selected digital gate where it cascades with the preceding logical products, until the final result is calculated at V_o after as many cycles as the analog operations. **b** An example query with 3 operations applied to the 41×303 crossbar programmed according to the heart-disease database values. The final operation C = A AND B, is using the partial results A, B that correspond to OR operations. A simple illustration of the cascaded logic system in the “product-of-sums” configuration that outputs the result C after the end of cycle #2. **c** Simulated waveforms of the digital circuitry solving an 11-step cascaded database query. The 3 nodes correspond to one column periphery and their positions are marked in **a**. The final result can be read at V_o node right after the end of the last clock cycle (#6). **d** Simulated computational metrics for solving the 11-step example query on the heart disease-related database.

4. Matching with automata processor

In this section, we will explore the applications of pattern matching. These applications are modelled using finite-state automata (FSA). The execution of FSA can be accelerated by a special form of CIM architecture, namely the automata processor.

4.1 Motivation

Pattern matching is widely used in diverse fields, including network security, computational biology, and data mining. This type of applications is challenging since they may involve a large amount of data. In this section, we present PROTOMATA[1] (for PROTein autOMATA) as an example application to illustrate its usefulness and challenges.

Each protein consists of a linear sequence of amino acid. There are 20 amino acids that occur naturally in nature, and each can be represented with a capital letter. In this way, we can present a protein as a letter string. PROTOMATA inspects given protein sequences for the occurrences of specific patterns. These protein patterns are called *motifs*, which play a biologically meaningful role. Recognizing these patterns are crucial for the researchers to understand the feature of the given protein sequence.

Figure 4 shows an example of a matched pattern in a protein sequence. The pattern, shown on the top of the figure, is taken from PROSITE[2] and follows their notation style. In this style, ‘<’ denotes the beginning of the sequence, ‘x’ any amino acid, ‘(10, 115)’ a repetition between 10 and 115 times, ‘−’ concatenation, ‘[...]’ a character class, and ‘{...}’ a complementary class, i.e. any amino acid but the ones listed within the curly braces. The protein sequence is named as Lissencephaly-1 homolog (D3BUN1), acquired from the UniProt database[3]. The first 115 amino acids in the sequence is underlined. They are followed by “F”, which is in the class of [DENF]. Following “F”, the amino acid “S” also matches the expected class [ST]. Similarly, the following string “VIVSAS” matches the remaining part of the pattern.

< x(10, 115) − [DENF] − [ST] − [LIVMF] − [LIVSTEQ]−V −{AGPN}−[AGP]−[STANEQPK]

```
MVLTNKQKEE LNGAILDYFD SSGYKLTSTE FTKETNIELD PKLKGLLEKK
WTSVIRLQKK VMDLEAKVSQ LEEELNNGGR GPARRGKEDA LPRQPEKHVL
TGHRNCINAV RFHPLFSVIV SASEDATMRI WDFDSGDFER TLKGHTNAVQ
DIDFDKSGNL LASCSADLTI KLWDFQSFDC IKTLHGHDHN VSCVRFLPSG
DQLVSSSRDK SIKVWETATG YCTKTLTGHE DWVRKVIVSE DGTTLASCSN
DQTARVWNLA KGECLLTFRE HSHVVECLAY SPANIVEVPG SLLSTPEGKA
KAKAGAGGTS FGQAGYLATG SRDKTIKIWE LATGRCLQTY IGHDNWVRSI
KFHPCGKYLI SVGDDKSIRV WDIAQGRCIK TINEAHSHFI SCLDFCSHNP
HIATGGVDDI IKIWKLG
```

Figure 4. Occurrence of the PROSITE motif PS00430 in the Lissencephaly-1 homolog (D3BUN1) protein

[1] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru, “High Performance Pattern Matching Using the Automata Processor,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 1123–1132, doi: 10.1109/IPDPS.2016.94.

[2] https://prosite.expasy.org/

[3] https://www.uniprot.org/

Many protein motifs have been discovered. For example, PROSITE has collected 1311 motifs by the time of April 2020. To match a protein sequence with all the pattern using conventional computer architectures requires a long execution time and large energy consumption.

4.2 Implementation with CIM

PROTOMATA implements the pattern matching using FSA, similarly to many other applications. If we can accelerate the execution of general FSA, then we can accelerate many pattern matching applications.

An FSA can be represented using a 5-tuple: (Q, Σ, δ, $q0$, C). Q denotes a set of states, Σ is a set of possible input symbols, δ is a function describing the set of possible transitions among the states, $q0$ is one of the states from Q and presents the start state, and C is a subset of Q and contains the accepting states. Note that each transition in δ has the form of $Q \times \Sigma \rightarrow 2^Q$, where 2^Q is the power set of Q, i.e., the set of all the subset of Q. An automaton processes an input symbol sequence and produces a Boolean value A that indicates whether the input sequence is accepted. The processing is done by altering the set of active states P based on the input symbol and δ. If $P \cap C \neq \varnothing$, then the input sequence is accepted.

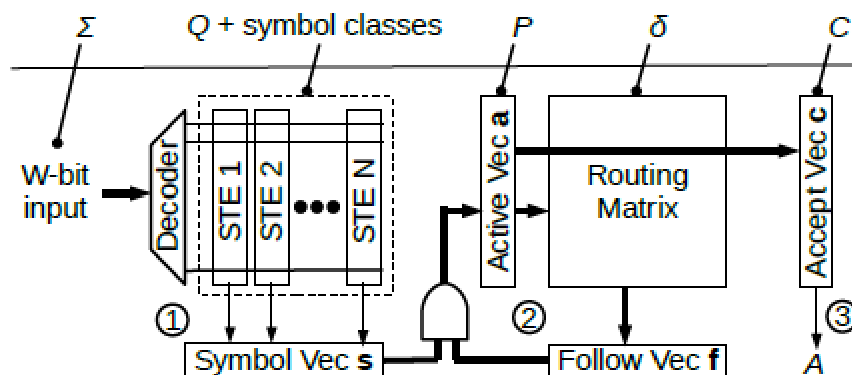


Figure 5. Structure of automata processor

An architecture named automata processor[4] is proposed to accelerate FSA using customized memory chip. It can be represented by the diagram shown in Figure 5. In every clock cycle, an input symbol I is processed using three major steps:

1. *Input symbol matching*: All the states that have incoming transitions occurring on I are identified in this step. Formally, this step identifies the union of b, where for any $a \in Q$ that $a \times I \rightarrow b \in \delta$. The N states are presented by column vectors called state-transit elements (STEs) which are pre-configured based on the targeted automaton. The decoder activates one of the word lines according to the input symbol I. If an STE has an incoming transition occurring on I, its corresponding output is logic 1;

[4] Dlugosch, P., D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing." *IEEE Transactions on Parallel and Distributed Systems*, IEEE Transactions on Parallel and Distributed Systems, 25, no. 12 (December 2014): 3088–3098. https://doi.org/10.1109/TPDS.2014.8.

otherwise, it is logic 0. The outputs of all STEs are mapped to a vector called Symbol Vector *s*.

2. *Active state processing*: It generates all the possible states that can be reached from the currently active states (stored in Active Vector *a*) based on the transition function (stored in the switching network), and stores the result in the Follow Vector *f*.
3. *Output identification*: Accept Vector *c* is pre-configured based on the automaton's accepting states. This step checks the intersection of *a* and *c* to decide whether the input sequence is accepted.

The STEs and the routing matrix are both implemented with memory arrays, and in particular, they can be implemented using memristive arrays such as RRAM. We refer to this design as *RRAM-AP*, and its chip structure is shown in Figure 6. As the STE matrix can be huge, it is fragmented across the entire chip and we refer to each fragment as a *tile*. RRAM-AP uses a hierarchical switching network that consists of global and local switches to implement the routing matrix. If the communication takes place inside a tile, only local routing is used; otherwise, global routing is used as well. In the figure, the Active Vector *a* is divided into several groups. Each group has some signals that enter global switches (represented by the box *G* in the figure) which are used for inter-tile communication. The outputs of the global switches combined with the initial vector *a* forms a vector (referred to as Global Vector *g*) and is used as the input to the local switches, which are presented by boxes *L1*, *L2*, and *L3*. The outputs of local switches form the Follow Vector *f*.

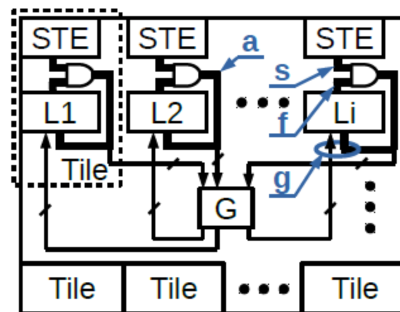


Figure 6. Chip structure of RRAM-AP

Figure 7 shows the implementation of STEs and routers in RRAM-AP. In Figure 7(a), each column represents an STE. It generates a bit in the Symbol Vector *s* based on the input symbol. The black and white boxes represent different configuration bits, and the triangles represent sense amplifiers. In Figure 7(b), the memory array is used as a part of the routing matrix, which is called a router. Figure 7(c) shows the detailed structure (1T1R) of a configuration bit. The bit line is pre-charged before evaluation, and the word lines are selected, e.g., by the input symbols. Note that for the routing matrix, multiple word lines can be activated in parallel. The vector dot product is calculated when all the word lines are set; if all the corresponding selected cells contain a high resistance (i.e., logic 0), then the pre-charged bit line remains high, and the sense amplifier (SA) will read a logic 0 (inverted output). Similarly, if at least one of the cells contains a low resistance (i.e., logic 1), then BL will be discharged. The SA's output will subsequently be a logic 1.

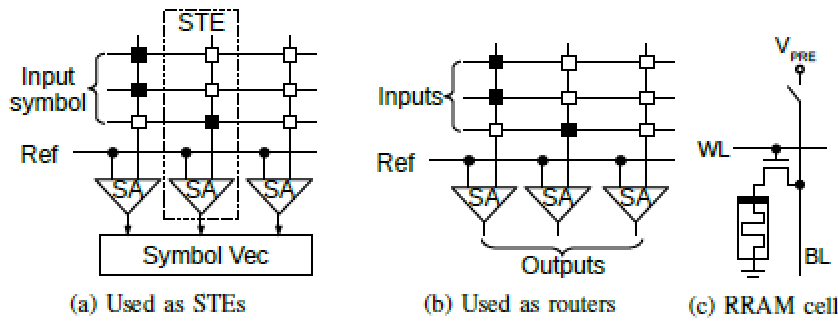


Figure 7. STEs and routers implementation with RRAM arrays

The characteristics of memristive devices provide opportunities for better implementation than conventional memory technologies. For example, an SRAM-based design named Cache Automaton[5] uses eight transistors to implement the configurable bit, whose area is much larger than the 1T1R structure. In addition, the SRAM cells also suffer from leakage power. As RRAMs are non-volatile devices, RRAM-AP can resume the last configured FSA after shut down and reboot without reprogramming it. On the other hand, RRAM-AP also inherits some drawbacks, such as the longer and power-hungry programming phase, and lower endurance, in comparison with DRAM and SRAM.

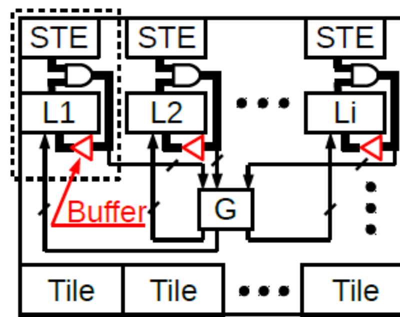


Figure 8. Switching network in RRAM-AP and TDM-AP

As the global switches are used to form an interconnection between the different tiles, they suffer from long global wires. They affect the latency of the *active state processing* step (Step 2) as it is determined by the sum of the latency of global and local switches. It is the performance bottleneck of RRAM-AP. By inserting buffers between the global and local switches, as shown in Figure 8, we can change the switching network into two pipeline stages and further improve the working frequency of the chip. However, due to data dependency, the automata states can no longer be updated within a cycle. To guarantee the processing correctness and fully utilize the hardware, multiple input streams enter the chip in a time-division multiplexing (TDM) manner. We refer to this design as *TDM-AP*. A multiplexer and a de-multiplexer are added to process the input and output signals. They can be controlled by the same selection signal with two additional buffers.

[5] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache Automaton," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, Cambridge, Massachusetts, 2017, pp. 259–272, doi: 10.1145/3123939.3123986.

4.3 Evaluation based on simulations

We conducted a SPICE simulation to estimate the maximum working frequency of TDM-AP. The simulation setup is listed in Table 3. The RRAM parameters are set following a $Pd/Al_2O_3/HfO_2/NiO_x/Ni$ RRAM device. To simplify and speed up the simulation, only one complete row and column of the STE arrays, global, and local switches are simulated. In such columns, only one cell is configured to low resistance. During the computation of an inner product, this configuration results in the highest discharge time, and therefore, it determines the minimum clock period. To guarantee a correct sense amplifier output, we need to make sure that the difference between the bit line and reference voltage V_{Ref} is larger than ΔV_{min}, which is the minimum voltage difference that the sense amplifier requires to operate correctly. When the RRAM cells in a column are all configured as logic 0, the voltage drop in the bit line is negligible due to the high resistance of the RRAM devices. As a result, we set V_{dd} = 1.1 V, V_{Ref} = 0.95V, and ΔV_{min} = 150 mV as shown in Table 3. With respect to the latency of global wires, we follow the assumption of Cache Automaton. Therefore, the latency introduced by the global wire is 99 ps.

Table 3. SPICE Simulation Setup

	Parameter	Value
RRAM Model: ASU	Top electrode width	40 nm
	Bottom electrode width	80 nm
	High resistance	10^9 Ω
	Low resistance	10^3 Ω
CMOS Model: TSMC	Technology node	40 nm
	V_{dd}	1.1 V
	V_{Ref}	0.9 V
	ΔV_{min}	150 mV
Global wire	Pitch	1 µm
	Length	1.5 mm
	Latency	66 ps/mm

Figure 9 shows the simulation result of an operation in the local switch, i.e., the inner product between the Global Vector *g* and *a* configuration vector. The bit line is first pre-charged to V_{dd}, which is controlled by the active-low signal Precharge. Then, *g* is used to activate the word lines. As a result, the bit line starts to discharge as one cell has a low resistance path. After a while, the sense amplifier is enabled, and it finally generates a positive output. The period between the rising edges of *g* and the sense amplifier's output is the latency of the local switch; it is approximately equal to 178 ps.

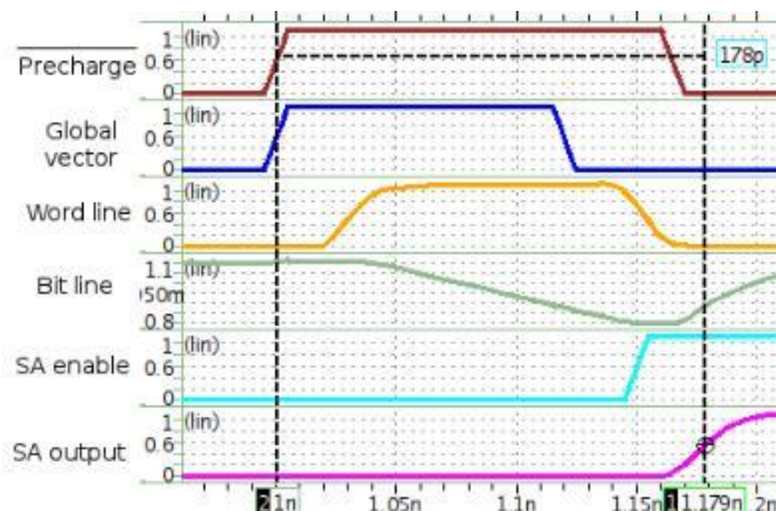


Figure 9. The waveform of SPICE simulation regarding TDM-AP

Similarly, other simulation shows that the latency of an STE array, an AND gate, a global switch, and a 64-to-1 OR gate are 258 ps, 11 ps, 129 ps, and 32 ps, respectively. Therefore, the latency of each step can be decided:

1. Input symbol matching: 258 ps.
2. Active state processing: This step consists of two phases. Global switching phase: 11 +99 +129 =239 ps. Local switching phase: 99 +178 =277 ps.
3. Output identification: 178 +32 =210 ps

Note that the three steps work in parallel. The step or phase that has the largest latency decides the minimal clock period of TDM-AP. With the above simulation result, we conclude that it is the local switching phase, whose latency is 277 ps. Therefore, it is safe to assume that TDM-AP can work at a frequency of 3.0 GHz. In each clock cycle, TDM-AP processes one 8-bit input symbol, which leads to a throughput of 24.0 Gbps.

With above simulation result, we can estimate the performance improvement on PROTOMATA introduced by TDM-AP. The protein motif matching conducts slowly using conventional CPUs. To process 18 million sequences that contain 6.58 billion amino acids in total, software matching needs 942,741 s while PROTOMATA executes for only 912 s on a DRAM-based automata processor. Among the execution time of PROTOMATA, the streaming time is 51.436 s, corresponding to a throughput of 1.024 Gbps. The rest time is to handle the matched output. Since we focus on the pattern matching part of the application, we only compare the matching throughput with CIM implementation.

Table 4. Throughput Comparison Among State-of-the-art Automata Accelerators

Designs	Frequency	Throughput (Gpbs)
DRAM-AP	125 MHz	1.024
HARE (w=32)	1.0 GHz	3.9
UAP	1.2 GHz	5.3
Cache automaton	2.0 GHz	15.6
TDM-AP	3.0 GHz	24.0

Table 4 lists state-of-the-art hardware accelerators for FSA and their working frequency and throughput. HARE is an ASIC specially designed for automata processing. UAP contains

multiple simplified cores. We can see that the throughput of TDM-AP is much higher than the others, including the DRAM-based automata processor. Therefore, it can accelerate PROTOMATA significantly.

5. Image processing application

One of the main goals in Mnemosene is to reduce the data transfer between the memory organisation and the processor cores. An important task in that data transfer is related to the address generation. Normally each packet of data that moves between a processor and a memory in a general processor SoC (Figure 10) contains 3 important parts:

Instruction	Target word address	Operands

For example, in a conventional system to write in a line of memory, we provide the target word address to be written. Then instruction is the “write instruction” and the operand is the “write data”. When performing an in-memory process, it is possible to give higher-level instructions to the memory block. For example, an instruction can include a binary AND between the Operand and the content of the target word address.

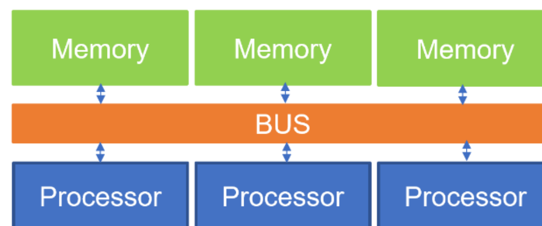


Figure 10: A general processing SoC

The overhead of the “target address word” in every transaction can be considerable. Address bits scale up with the size of the memory and adds considerable overhead to each memory transaction. In imec, we have come up with an innovative scheme (submitted for patenting) to reduce the number of transactions required for address transfer by the implementation of a more flexible hardware-supported scheme that can generate a complex pattern of addresses from a packed address instruction.

Even though this scheme is more advanced than just generating a sequence of addresses, it is implemented with lightweight digital sequential shift registers. This design is called the address calculation accelerator (ACA) and it is categorized as CiM-P architecture. ACA can generate sequences both in rows and columns when multiple works are stored in one row. Additionally, it is possible to select only part of a word (like masking) when required.

This process is very useful when a sequence of addresses in consecutive transactions follows a repetitive pattern and when in every iteration, we have to perform similar instructions. In this case, many small transactions can be packed in the following format:

Instruction	ACA Operands	Operands

ACA operands are used by the ACA unit to unpack the sequence of addresses. This means the instructions compiled in the processor should be packed using an additional ACA compiler.

The target domain for this approach are all streaming applications which have regular or partly regular addressing schemes. So these are mainly loop kernel oriented but some conditional control flow is allowed. Hence, this corresponds mostly to the GPU target domain.

To demonstrate the performance of this scheme we have collaborated with the IPI group of Prof. Wilfried Philips and Prof. Bart Goossens at UGent&IMEC, Belgium. They work on advanced image processing algorithms and together we have chosen a representative image processing technique called “guided image filtering”. This application uses two images as input and guide and performs repetitive operations on the input image using the guided filter. In our case, the sizes of the input image, guided filter, and output are the same.

This application follows the following pipeline to process an input image:

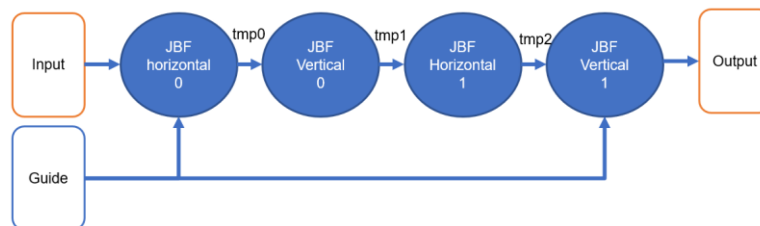


Figure 11: The pipeline of the guided image filtering application. JBF stands for “Joint Box Filter”

To demonstrate ACA performance as part of the Mnemosyne project, we have implemented a prototyope ACA compiler that starts from a code sample and which compresses the access patterns to the memory and encodes it with the ACA type instruction packet. This compiler is still in a preliminary shape and will be finalized in the future work. Additionally, we have implemented the ACA hardware circuit in HDL code to extract power/throughput performance and we have included it in our WP4 nano-simulator (see WP4.2&4.3). To demonstrate the feasibility for use in emerging CIM memories, we have coupled it to the memory models for our imec STT-MRAM technology (seeWP4 again).

Even though we believe that also the arithmetic image pixel operations required for this application can be accelerated using other CiM-A and CiM-P methods available in MNEMOSENE, our concern here is about the large amount of address transactions present in the guided image filter code.

As a first attempt, we have tried to execute the pipeline stages of this application serially. This means, at every moment in time, ACA was involved in processing one of the kernels (tmp0/tmp1/tmp2/output). As mentioned before, ACA can reduce the number of individual transactions over the BUS by packing/unpacking the addresses. In these experiments, we measured the number of individual transactions before and after using ACA compression. Please note that we only compress the address/instruction fields and operands are required to be transferred in the packet without any compression. Additionally, we run the experiments for different input sizes, as it affects the compression ratio.

The following table shows the results of ACA compression for an image size of 32×32.

Image size: 32×32			
Kernel	Normal	Compresses	Ratio
Input	33793	5953	18%
Guide	36865	5954	16%
Tmp0 (Hor0)	147457	5954	4%
Tmp1 (Ver0)	73729	33730	46%
Tmp2 (Hor1)	73729	5954	8%
Output (Ver1)	3073	1	0.03%
Total	**368646**	**57546**	**16%**

For Tmp1, we noticed that with a simple modification of the algorithm, we can reach a higher compression ratio as shown in the following table.

Image size: 32×32			
Kernel	Normal	Compresses	Ratio
Input	33793	5953	18%
Guide	36865	5954	16%
Tmp0 (Hor0)	147457	5954	4.0%
Tmp1 (Ver0)	73729	5954	8.0%
Tmp2 (Hor1)	73729	5954	8.0%
Output (Ver1)	3073	1	0.03%
Total	**368646**	**29770**	**8%**
Image size: 256×256			
Kernel	Normal	Compresses	Ratio
Input	2162689	219649	10%
Guide	2359297	219650	10%
Tmp0 (Hor0)	9437185	219650	2.3%
Tmp1 (Ver0)	4818593	219650	4.5%
Tmp2 (Hor1)	4718593	219650	4.6%
Output (Ver1)	196609	1	0%
Total	**23692966**	**1098250**	**4.6%**

In this execution method, we process each kernel sequentially. However, as it is clear from Figure 11, it is possible to execute them in parallel by exploiting a software pipelining concept. As all the kernels execute similarly with the unique access pattern to the memory, in the parallel execution, we can use a longer word line in the memory to feed all the processes in parallel. The outcome is shown in Figure 12.

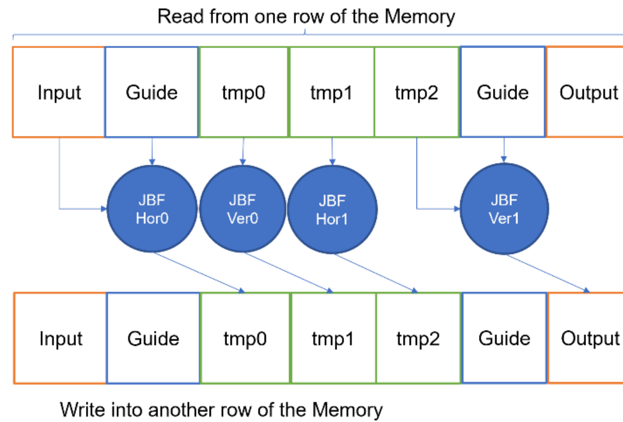


Figure 12: Parallel read/write from a wide memory in guided image filtering

In the parallel processing form, a long word of the memory is read, processed, and write back to the memory. In this way, we save even more in address transactions. Processing one long word can take one or several cycles, dependent on the target architecture. In this case, ACA only needs to generate one address per line which results in a reduced cycle count and hence a higher performance and also energy efficiency for the address generation and the address and data communication network. However, the energy consumption for the memory access itself and the arithmetic instructions on the processor cores mainly remains the same. So, the biggest gains are expected on the overall throughput and latency combined with a medium gain on the total energy consumption.

Image size	Normal	Compresses	Ratio
32x32	368646	5958	1.6%
256x256	23692966	219654	0.92%

ACA can be used with any embedded memory technology (SRAM, STT, SoT-MRAM, eDRAM, HBM, ...) and arithmetic processing architecture (CiM-A, Near memory, ...).

6. Deep learning inference application

In this section we describe the improvement of the HW-SW Co-design framework for analog DNN accelerators for IoT applications using NVM crossbars presented in D1.2. Therefore, this work is motivated by the studies presented in the deliverable D1.1, and whose initial results were reported in D1.2.

In this work, we revise the previously presented training framework for improving HW blocks re-usability. In the previous framework version the quantization algorithm obtained uniform scaling across the DNN layers independently of their characteristics, removing the need of per-layer full-custom design while reducing the peripheral HW. In this iteration of the framework, we propose a training algorithm that allows the NN (or part of it) to be mapped to only-positive weights, leading to important energy and area savings.

6.1 Motivation: Challenges Related to Weights Polarity in Crossbar NN Accelerators

The conductance in a passive NVM element can only be a positive number g in the range $[g_{OFF}, g_{ON}]$. However, the NN weights, no matter whether W is a real or an integer number, W contains both positive and negative values. Consequently, the use of bipolar weights involves a problem when mapped to an only positive conductance set.

Traditionally positive and negative weights are deployed separately in different areas of the crossbar. This approach comes with the duplication of crossbar area and energy consumption, and the addition of current subtractors or highly-tuned differential ADC hindering the reconfigurability of the accelerator. As depicted in the next figure, using this scheme, we double the crossbar area as per-weight, one column computes the positive contributions, while the other column the negative ones.

Moreover, additional current subtraction blocks are required before/at the ADC stages [13] [14]. Alternative solutions as [15] shifting the weight matrices usually involve the use of biases dependent on the inputs and additional periphery. Nevertheless, both alternatives involve considerable area and energy overheads.

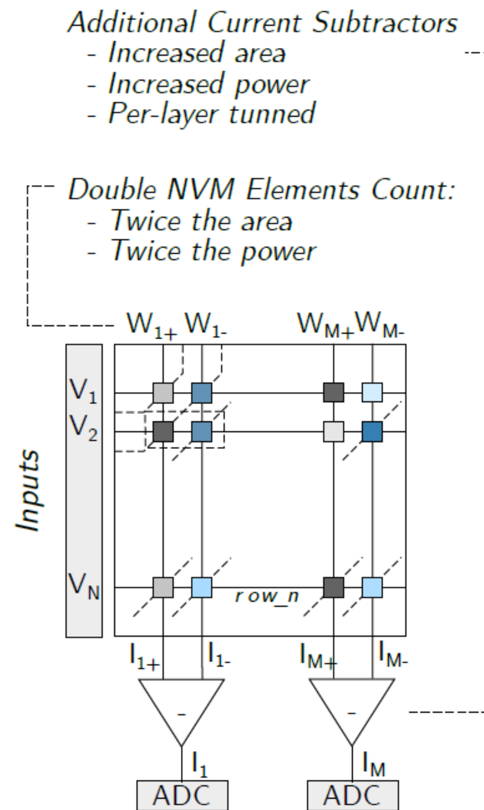


Figure 13 Problems found on the deployment of bipolar weights in crossbars

6.2 Hard-Constrained Quantized Training

The present section summarizes the hard-constrained quantized training algorithm presented in D1.2. The quantization of both weights and activations is a critical step on the design of the accelerated system defining the system accuracy, area and power consumption. To avoid per-layer scaling and thus enabling system reconfigurability while reducing the area/power resources, a HW-SW co-design stage is required at training time. We refer the reader to D1.2 for further details.

Our methodology proposes the use, for all the hidden layers present in the NN, of:

1. a single ADC design performing ***act()***
2. a single DAC design performing ***to_v()***
3. a single weight mapping function ***f()***
4. a global set of activation values ***Y_g = [y_0, y_1]***
5. a global set of input values ***X_g = [x_0, x_1]***
6. a global set of weight values ***W_g = [w_0, w_1]***,

and being the crossbar behavior defined by

$$
\begin{aligned}
i_i &= \sum v_{ik} g_{ikj} \\
v_{ik} &= to_v(x_{ik}) \\
g_{ikj} &= f(w_{ikj}) \\
y_{ij} &= act(i_{ij}).
\end{aligned}
$$

To achieve the desired behavior we need to ensure at training time that the following equations are met for each hidden layer *L_i* present in the NN:

$$
\begin{aligned}
Y_i &= \{y_{ij}\}, y_{ij} \in [y_0, y_1] \\
X_i &= \{x_{ik}\}, x_{ik} \in [x_0, x_1] \\
W_i &= \{w_{ikj}\}, w_{ikj} \in [w_0, w_1].
\end{aligned}
$$

In most cases, but more commonly in classification problems the output activation (sigmoid, softmax) does not match the hidden layers activation. Therefore for the DNN to learn the output layer should be quantized using an independent set of values ***Y_o, X_o, W_o*** that may or not match ***Y_g, X_g, W_g***. Consequently, the output layer is the only layer that once mapped to the crossbar requires full-custom periphery.

Our framework translates the previously defined variables, signals and functions into the following graph

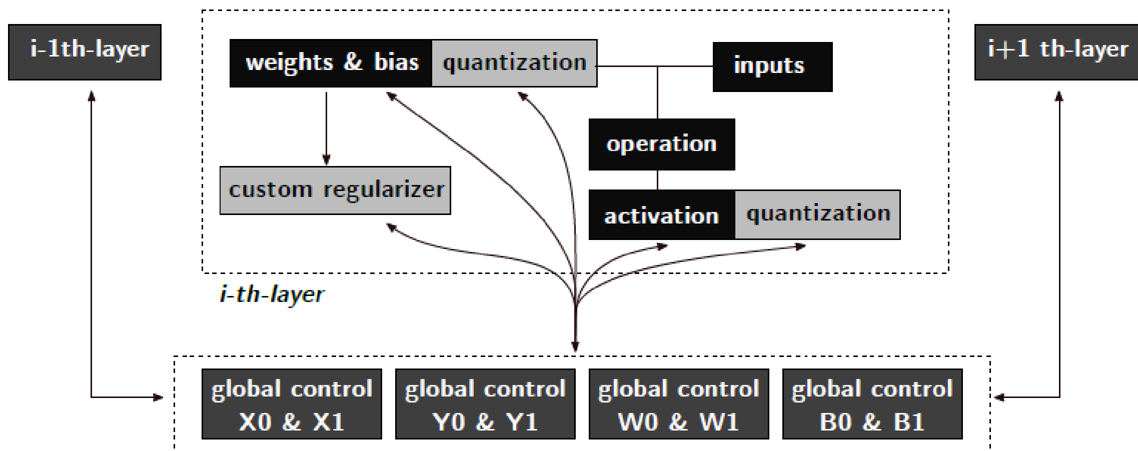


Figure 14 HW aware hard constraint training graph

6.2.1 Loss Definition

Our target of achieving complete (or blocks of the) NN being mapped to unipolar (only positive may lead to non-convergence issues. In order to help the convergence towards a valid solution, we introduce extra L_C terms in the loss computation that may depend on the training step.

The final loss $\boldsymbol{L_F}$ is then defined as

$$\mathcal{L}_F = \mathcal{L} + \mathcal{L}_{L2} + \mathcal{L}_{L1} + \mathcal{L}_C,$$

Where ***L*** refers the standard training loss, $\boldsymbol{L_1}$, $\boldsymbol{L_2}$ refer the standard L1 and L2 regularization losses, and $\boldsymbol{L_C}$ is the custom penalization. An example of this particular regularization terms may refer the penalization of weight values beyond a threshold $\boldsymbol{W_T}$ after training step ***N***.

This loss term can be formulated as

$$\mathcal{L}_C = \alpha_C \sum_w max(W - W_T, 0) HV(step - N)$$

where $\boldsymbol{alpha_C}$ is a preset constant and ***HV*** the Heaviside function. If the training would still provide weights whose values surpass $\boldsymbol{W_T}$, HV function can be substituted by a non-clipped function ***relu(step-N)***. In particular, this $\boldsymbol{L_C}$ function was used in the unipolarity experiments located at the results section.

6.2.2 Unipolar Weight Matrices Quantized Training

Mapping positive/negative weights to the same crossbar involve double the crossbar resources and introducing additional periphery. Using the proposed training scheme we can restrict further the characteristics of the DNN graph obtaining unipolar weight matrices, by redefining some global variables as

$$W_g \in [0, w_1]$$

and introducing the $\boldsymbol{L_C}$ function defined in the previous section.

Moreover, for certain activations (***relu, tanh***, etc.) the maximum and/or minimum values are already known, and so the sets of parameters in ***Vg*** can be constrained even further. These maximum and minimum values can easily be mapped to specific parameters in the activation function circuit interfacing the crossbar.

Finally, in cases where weights precision is very limited (i.e. 2 bits), additional loss terms as $\boldsymbol{L_C}$ gradually move weight distributions from a bipolar space to an only positive space, helping the training to converge.

In summary, by applying the mechanisms described, we open the possibility of obtaining NN graphs only containing unipolar weights.

6.3 Experiments and Results: Unipolar Weights vs Accuracy Trade-off

Following the work presented in D1.2, we have evaluated the presented methodology using CIFAR10 and Human Activity Recognition (HAR) applications. CIFAR10 comprises the classification of 32x32 sized images into 10 different categories. HAR classifies among incoming data from different sensors (accelerometer, gyroscope, magnetometer, 3 channels

each) into 12 different activities (run, jump, etc.) To mimic a smartwatch scenario we used real data from sensors placed in only one limb from [16] dataset.

In the present section, the evaluated NN were the same as the reported in D1.2, where only the graph elements to achieve unipolarity were added.

6.3.1 Fully Connected DNN: HAR

In this experiment we apply the proposed mechanisms to obtain a NN classifying different HAR activities whose weights take only positive values.

The Differentiable Algorithm (DA) conducted the exploration of the NN design space, determining the NN architecture and parameters set that provided the best accuracy while using only positive weights within the NN. To help the NN training to converge, and following graph structure shown in the previous section, custom regularizers were required to penalize negative weights, and a variation of alpha-blending quantization scheme [17] was introduced.

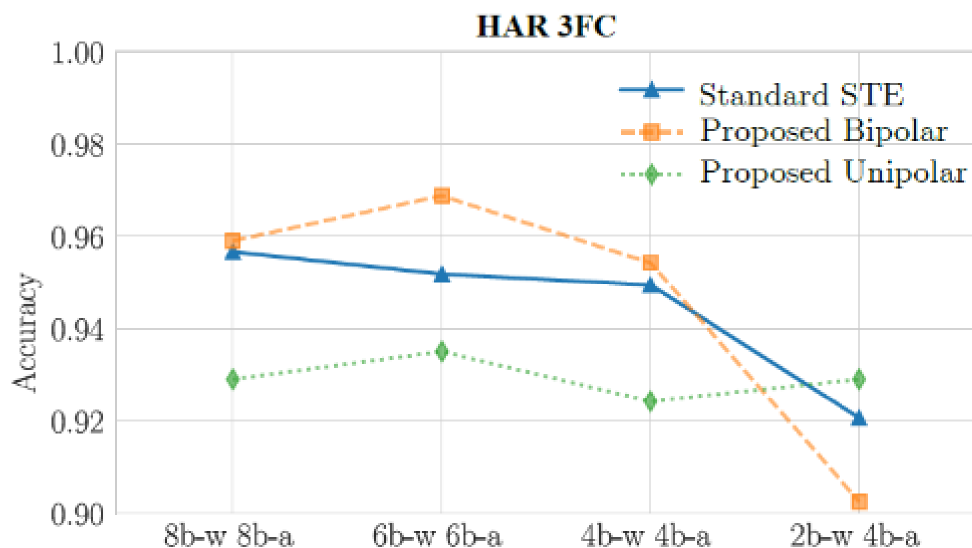


Figure 15 Accuracy study on HAR application varying different quantized training techniques with different precision.

The figure above summarizes the experiment results. NNs with bipolar weight matrices use ***relu*** as the hidden-layers activation. On the contrary, our design exploration algorithm found ***act = tanh(x - th_g)*** as a function best suiting NNs based on unipolar weight matrices. The introduction of ***th_g*** shift on the ***tanh*** function allows the network to map a small valued positive (negative) input to the activation as a small valued negative (positive) at the output, .

The proposed solution is as competitive as the standard one, while obtaining the significant benefit of a reduced set of weights and uniform HW. But more importantly, we demonstrate that small NN using only unipolar weight matrices/ADCs can correctly perform classifications, aiding the deployment to NVM crossbars.

6.3.2 Deeper CNN: CIFAR10

For larger convolutional networks the imposed unipolarity constraint can be too restrictive for the NN to correctly learn. We propose imposing the constraint only to a certain number of channels in each layer. The ratio of unipolar/bipolar channels will determine the final accuracy and the power and area savings.

The following Figure describes the results of applying hard unipolar weights constraint to the same CIFAR10 application, varying the percentage of unipolar channels in each convolutional stage from 0% (bipolar weights) to 100% (completely unipolar weights), for the standard STE and uniform-scaling quantization approaches.

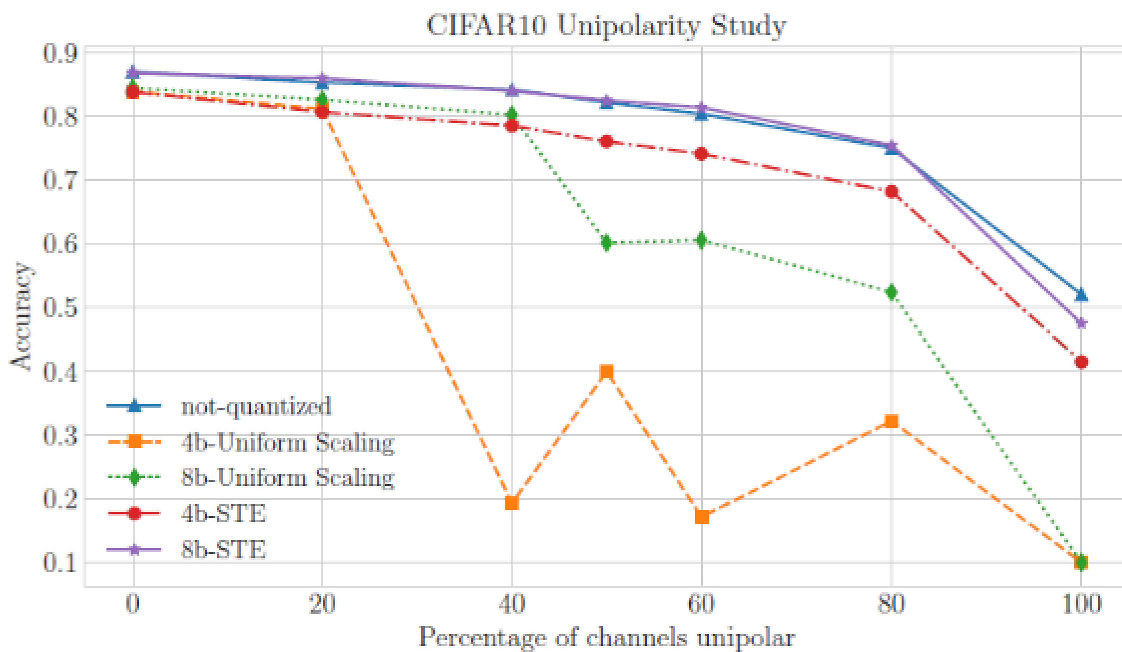


Figure 16 Unipolarity study on larger CNNs. Accuracy versus percentange of channels constraint as unipolar.

It can be seen how a minimum number of channels in each convolutional layer is required by the NN to learn. Unipolar percentages above 60% impose a hard limitation, especially when the uniform-scaling training scheme is used.

However, it can be seen how for the 8-b STE quantization scheme, by imposing 50% unipolar channels, we can reduce a 25% the crossbar area/energy with a small 2% accuracy penalty. For the 4-b scheme, a 20% area/energy savings would come with a 4% accuracy reduction. Therefore, we can state that even for more complex problems, we can greatly simplify the NN deployment forcing a percentage of channels to be unipolar.

6.4 Energy and Area Benefits

The following section shows the energy and area benefits obtained after applying our proposed methodology and framework to the experiments shown in both D1.2 and in the present document.

CIFAR10 NN numbers only refer to the bipolar experiments. Additional area/power improvements can be obtained should the user select a percentage of channels being unipolar, as shown in the previous section.

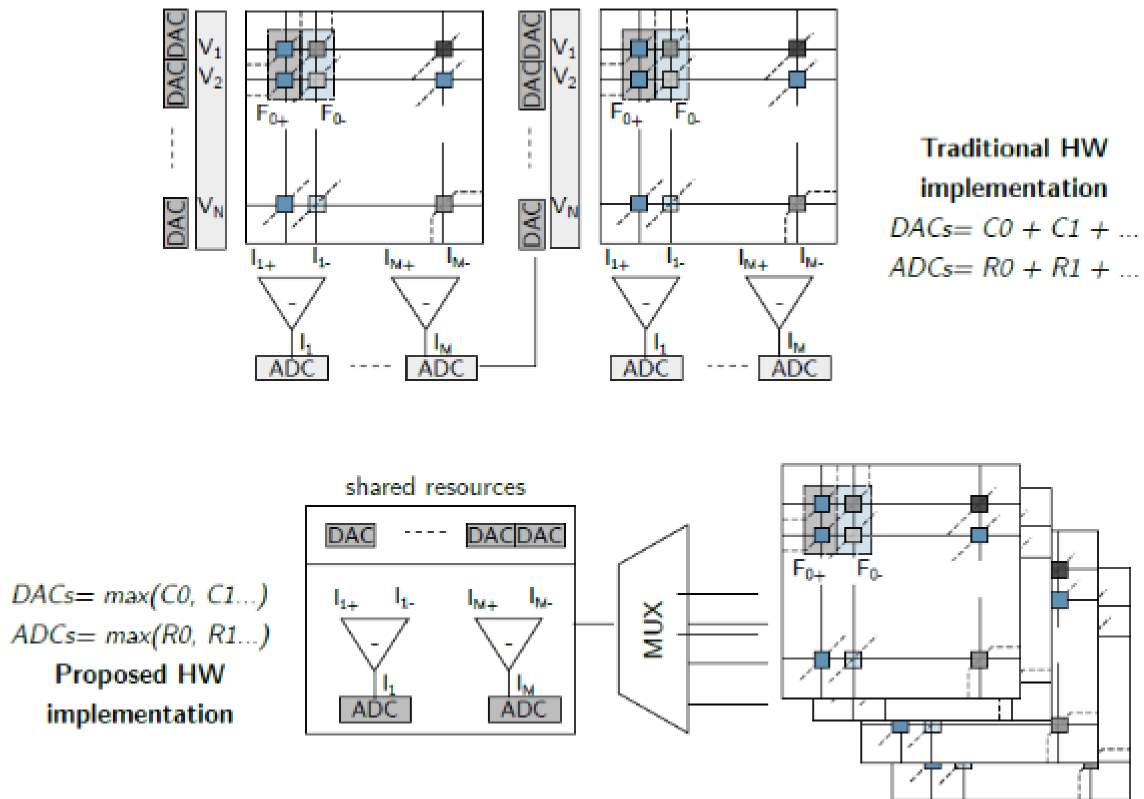


Figure 17 Comparison of traditional per-tile periphery versus proposed shared periphery.

The above figure describes the comparison of HW implementation, where we consider [18] numbers, where each PCM NVM element --each parameter in our NN-- consumes ~0.2 uW, and has a $25F^2$ area, equivalent to 0.075 um^2.

For the DAC/ADC characteristics, we designed in house 4-bit and 8-bit elements, using a 55nm CMOS technology. Simulated power consumption and area are gathered in the following table:

CHARACTERISTICS OF DESIGNED ADC AND DAC

Device	Power@10 MHz	@100 MHz	Area
DAC 4b	3.2 μW	11.7 μW	101 μm^2
DAC 8b	4.4 μW	13.6 μW	440 μm^2
ADC 4b	1.28 μW	12.56 μW	1030 μm^2
ADC 8b	1.64 μW	16.39 μW	7920 μm^2

A power/area overhead of 5/10 % over the ADC figure for an integrated adapted current subtractor is added in the case where bipolar weights are present. Additional 5% power penalty is applied for ADCs using current scaling. Regarding each one of the NN layers,

DACs and ADCs will only be multiplexed should the layer maintain uniform scaling with the system.

With our proposed approach, all layers share the same input ranges, and only the last layer ADCs would be different from the rest of the system.

6.4.1 Energy Estimation

To maximize the throughput per NN layer we consider one DAC (ADC) per column (row). From the power perspective, for each layer the total number of NVM cell reads performing the multiplications (and automatically the additions) would be

$$\sum_{F_i} \mathcal{K}_i \mathcal{X}_i$$

for the convolutional layers and

$$\mathcal{X}_i \mathcal{Y}_i$$

for the fully connected ones, where ***Xi, Yi, Ki*** refer the the size of inputs, outputs, and kernel respectively, and ***Fi*** refers the number of filters of the i-th layer. Regarding the DACs and ADCs utilization, a total of ***Xi*** and digital to analog and ***Yi x Fi*** analog to digital conversions are required. No analog scaling system is required. The results describing the power estimation per inference in both bipolar-CIFAR10 and bipolar/unipolar-HAR applications is displayed in the following table:

ESTIMATED ENERGY PER INFERENCE: NUMBER OF NVM CELL READS (POSITIVE (+) AND NEGATIVE(−) WEIGHTS), DAC/ADC OPERATIONS.

CIFAR10	**TF 8 bits 10/100 MHz**	**TF 4 bits 10/100 MHz**	**Ours 4 bits 10/100 MHz**
Total	1.6/0.19 μJ	1.59/0.18 μJ	1.58/0.18 μJ
NVM (±) ≈ 77e6	1.55/0.16 μJ	1.55/0.16 μJ	1.55/0.16 μJ
DAC ops ≈ 75e3	32.9/10.1 nJ	23.9/8.7 nJ	23.9/8.7 nJ
ADC* ops ≈ 115e3	22.6/22.6 nJ	18.4/18.1 nJ	16.5/16.2 nJ
HAR	**TF 8 bits 10/100 MHz**	**TF 4 bits 10/100 MHz**	**Ours 4 bits 10/100 MHz**
Total	1.6/0.24 nJ	1.54/0.22 nJ	0.84/0.15 nJ
NVM (+) ≈ 34e3	0.7/0.07 nJ	0.7/0.07 nJ	0.7/0.07 nJ
NVM (−) ≈ 34e3	0.7/0.07 nJ	0.7/0.07 nJ	0 nJ
DAC ops ≈ 384	0.17/0.05 nJ	0.12/0.04 nJ	0.12/0.04 nJ
ADC* ops ≈ 268	0.052/0.05 nJ	0.04/0.03 nJ	0.03/0.03 nJ

As bipolar weights were needed in the image solution, and due to the amount of multiplications (>38 million per inference), the saved power is almost negligible.

However, in very low power IoT applications, the proposed solution requires only 55% of the energy compared with traditional schemes, mainly due to the unipolar weight matrices encoded in the NVM crossbar.

6.4.2 Area Estimation

In traditional deployments, being ***Fi*** the number of filters present in a given layer ***Li***, ***Fi*** full custom different ADCs would be designed and placed for that layer, freezing the applicability to a particular application. However, with our proposed scheme we can deploy different NN applications in the same hardware, using many smaller and fixed-sized crossbars. We can feed the incoming inputs in batches, reusing the kernels unrolled in the crossbar.

Adopting this second scheme for the CIFAR10 example, the largest CNN unrolled layer requires an input of size 32x32x32. For example, if the crossbar size available in our reconfigurable system is 128x128, the layer can be batched in 256 operations. If the hardware blocks were composed of 512x128 elements, the layer could be batched in 64 operations. On the other hand, for smaller NN this same hardware could fit entire layers: in HAR benchmark each layer can fit in a 128x128 crossbar. For both crossbar size examples, every layer but the last would reuse the 128 DAC/ADC pairs during inference.

ESTIMATED AREA USING $128x128$ BASIC CROSSBAR BLOCKS.

CIFAR10	**TF 8 bits**	**TF 4 bits**	**Ours 4 bits**
Reconfigurable	No	No	Yes
Crossbars	44	44	44
DACs	448	448	128
ADCs	896	896	256
Current subtractors	896	896	256
Total Area	8.05 mm^2	1.1 mm^2	0.22 mm^2
HAR	**TF 8 bits**	**TF 4 bits**	**Ours 4 bits**
Reconfigurable	No	No	Yes
Crossbars	6	6	3
DACs	384	384	128
ADCs	268	268	256
Current subtractors	268	268	0
Total Area	2.51 mm^2	0.35 mm^2	0.28 mm^2

The table above summarizes the area estimation when considering crossbars composed of 128x128 elements (a very conservative approach to avoid technology problems) and assisted by 128 DACs, 128 ADCs and additional periphery. For the traditional approaches, we follow the deployment schemes in the literature, and consider that the number of ADCs present in each layer does not need to match the crossbar column size, saving considerable amount of area but avoiding reconfigurability.

On the contrary, by using the proposed solution the DACs and ADCs are multiplexed. The benefits are noticeable: First, we guarantee that the HW is uniform across the NN, ensuring reconfigurability. Second, in CIFAR10 benchmark, the solution leads to up to 80% area saving --0.22 mm^2 vs 1.1 mm^2 for 4-bit accelerators. For HAR benchmark, up to 20% area saving is achieved.

When comparing against the traditional 8-bit deployment schemes, this area saving raises up to 97% for CIFAR10 CNN benchmark and 89% for the HAR NN.

6.5 Conclusions

The work presented in the D1.2 and the present document a solution that aids the algorithm deployment in uniform crossbar/periphery blocks, at training time. With no accuracy penalty, the method can simplify the design of the crossbar periphery, significantly reducing the overall area and power consumption, and enabling real re-usability and reconfigurability.

Moreover, we have demonstrated that DNN with unipolar weight matrices can correctly perform bio-signals classification tasks while solving the negative/positive weights problem inherent to NVM crossbars, and therefore reducing by half the crossbar area/energy and significantly simplifying the periphery design. We validated our solution against two different always-ON sensing applications, CIFAR10 and HAR.

7. Hyperdimensional computing application

7.1 Implementation of HD computing with CIM

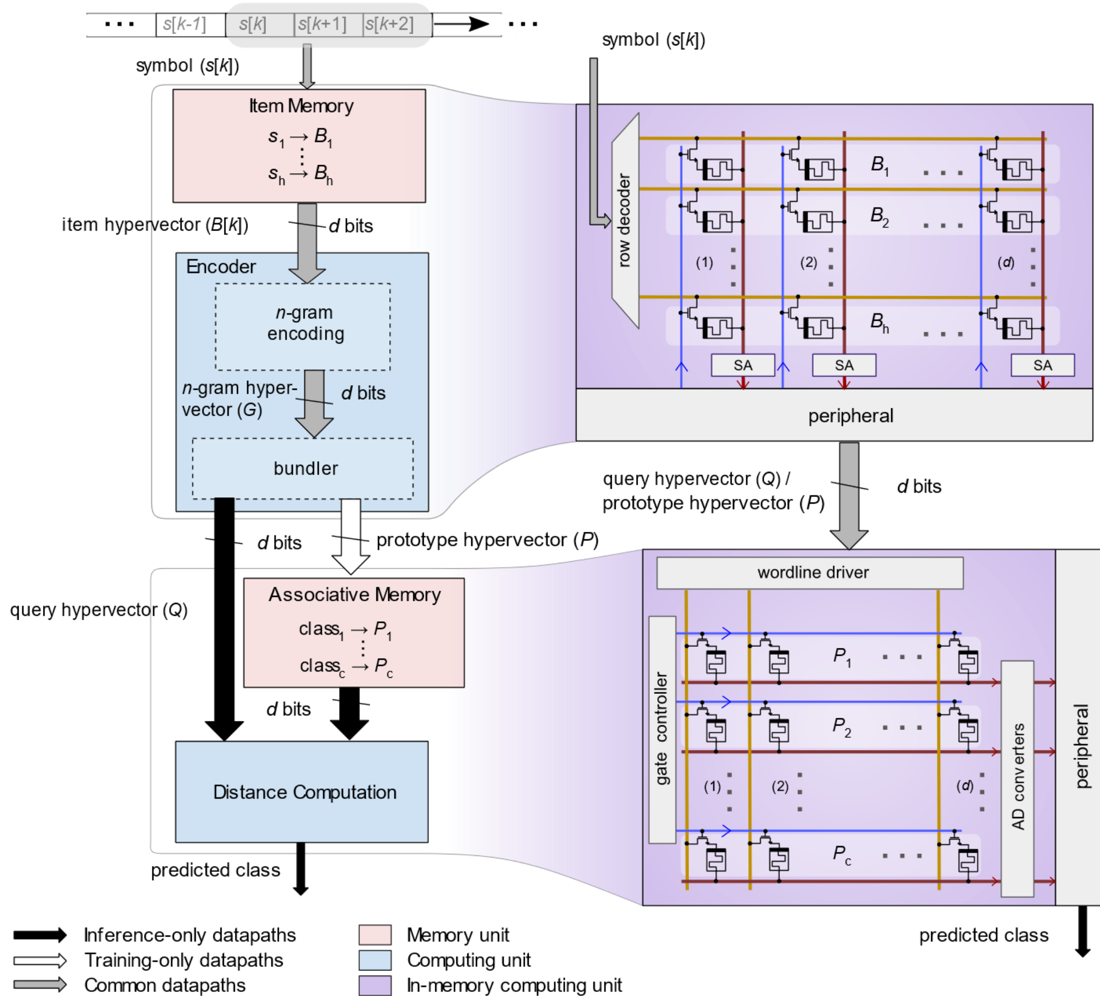


*Figure 18:**The concept of in-memory HDC**: A schematic illustration of the concept of in-memory HDC shows the essential steps. associated with HDC (left) and how they are realized using in-memory computing (right). An item memory (IM) stores **h**, **d**-dimensional basis hypervectors that correspond to the symbols associated with a classification problem. During learning, based on a labelled training dataset, an encoder performs dimensionality preserving mathematical manipulations on the basis hypervectors to produce **c**, **d**-dimensional prototype hypervectors that are stored in an associative memory (AM). During classification, the same encoder generates a query hypervector based on a test example. Subsequently, an associative memory search is performed between the query hypervector and the hypervectors stored in the AM to determine the class to which the test example belongs. In in-memory HDC, both the IM and AM are mapped onto crossbar arrays of memristive devices. The mathematical operations associated with encoding and associative memory search are performed in-place by exploiting in-memory read logic and dot product operations, respectively. A dimensionality of **d**=10,000 is used. SA: sense amplifier; AD converters: analog-to-digital converters.*

When Hyperdimensional Computing (HDC) is used for learning and classification, first, a set of i.i.d., hence quasiorthogonal hypervectors, referred to as basis hypervectors, are selected to represent each symbol associated with a dataset. For example, if the task is to classify an unknown text into the corresponding language, the symbols could be the letters of the alphabet. The basis hypervectors stay fixed throughout the computation. If there are ***h***

symbols $\{s\}_1^h$, the set of the ***h***, ***d***-dimensional basis hypervectors $\{B_i\}_1^h$ is referred to as the item memory (IM) (see Figure 18).

Basis hypervectors serve as the basis from which further representations are made by applying a well-defined set of component-wise operations: addition of binary hypervectors ***[+]*** is defined as the component-wise majority, multiplication $\oplus$ is defined as the component-wise exclusive-OR (or equivalently as the component-wise exclusive-NOR), and finally permutation (***ρ***) is defined as a pseudo-random shuffling of the coordinates. Applied on dense binary hypervectors where each component has equal probability of being zero or one [19] all these operations produce a ***d***-bit hypervector resulting in a closed system.

Subsequently, during the learning phase, the basis hypervectors in the IM are combined with the component-wise operations inside an encoder to compute for instance a quasiorthogonal ***n***-gram hypervector representing an object of interest [20]; and to add ***n***-gram hypervectors from the same category of objects to produce a prototype hypervector representing the entire class of category during learning. In the language example, the encoder would receive input text associated with a known language and would generate a prototype hypervector corresponding to that language. In this case ***n*** determines the smallest number of symbols (letters in the example) that are combined while performing an ***n***-gram encoding

operation. The overall encoding operation results in ***c***, ***d***-dimensional prototype hypervectors (referred to as associative memory (AM)) assuming there are ***c*** classes. When the encoder receives ***n*** consecutive symbols, $\{\mathrm{s}[1], \mathrm{s}[2], \ldots, \mathrm{s}[\mathrm{n}]\}$, it produces an ***n***-gram hypervector through a binding operation given by

$$G(s[1], s[2], \ldots, s[n]) = B[1] \overline{\oplus} \rho(B[2]) \overline{\oplus} \rho^{n-1}(B[n])$$

1

where ***B[k]*** corresponds to the associated basis hypervector for symbol, ***s[k]***. The operator $\overline{\oplus}$ denotes the exclusive-NOR (XNOR), and ***ρ*** denotes a pseudo-random permutation operation, e.g., a circular shift by 1 bit. The encoder then bundles several such ***n***-gram hypervectors from the training data using component-wise addition followed by a binarization (majority function) to produce a prototype hypervector for the given class.

When inference or classification is performed, a query hypervector (e.g. from a text of unknown language) is generated identical to the way the prototype hypervectors are generated. Subsequently, the query hypervector is compared with the prototype hypervectors inside the AM to make the appropriate classification. Equation 2 defines how a query hypervector ***Q*** is compared against each of the prototype hypervector P_i out of ***c*** classes to find the predicted class with maximum similarity. This AM search operation can for example be performed by calculating the inverse Hamming distance.

$$\mathrm{Class}_{\mathrm{Pred}} == \mathrm{argmax}_{i \in \{1,\ldots,c\}} \sum_{j=1}^{d} Q(j) \overline{\oplus} P_i(j)$$

2

One key observation is that the two main operations presented above, namely, the encoding and AM search, are about manipulating and comparing large patterns within the memory itself. Both IM and AM (after learning) represent permanent hypervectors stored in the memory. As a lookup operation, different input symbols activate the corresponding stored

patterns in the IM that are then combined inside or around memory with simple local operations to produce another pattern for comparison in AM. These component-wise arithmetic operations on patterns allow a high degree of parallelism as each hypervector component needs to communicate with only a local component or its immediate neighbours. This highly memory-centric aspect of HDC is the key motivation for the in-memory computing implementation proposed in this work.

The essential idea of in-memory HDC is to store the components of both the IM and the AM as the conductance values of nanoscale memristive devices [21], [22] organized in crossbar arrays and enable HDC operations in or near to those devices (see Figure 18).

The IM of ***h*** rows and ***d*** columns is stored in the first crossbar, where each basis hypervector is stored on a single row. To perform $\overline{\oplus}$ operations between the basis hypervectors for the ***n***-gram encoding, an in-memory read logic primitive is employed. Unlike the vast majority of reported in-memory logic operations [23], [24], [25] the proposed in-memory read logic is non stateful and this obviates the need for very high write endurance for the memristive devices. Additional peripheral circuitry is used to implement the remaining permutations and component-wise additions needed in the encoder. The AM of ***c*** rows and ***d*** columns is implemented in the second crossbar, where each prototype hypervector is stored on a single row. During supervised learning, each prototype hypervector output from the first crossbar gets programmed into a certain row of the AM based on the provided label. During inference, the query hypervector output from the first crossbar is input as voltages on the wordline driver, to perform the AM search using an in-memory dot product primitive. Since every memristive device in the AM and IM is reprogrammable, the representation of hypervectors is not hardcoded, as opposed to Refs. [26] [27] [28]. that used device variability for projection.

This design ideally fits the memory-centric architecture of HDC, because it allows to perform the main computations on the IM and AM within the memory units themselves with a high degree of parallelism. Furthermore, the IM and AM are only programmed once while training on a specific dataset, and the two types of in-memory computations that are employed, involve just read operations. Therefore, non-volatile memristive devices are very well suited for implementing the IM and AM, and only binary conductance states are required. In this work, we used PCM technology [29], [30], which operates by switching a phase-change material between amorphous (high resistivity) and crystalline (low resistivity) phases to implement binary data storage (see Methods). PCM has also been successfully employed in novel computing paradigms such as neuromorphic computing [31], [32], [33], [34] and computational memory [35], [36], [37], which makes it a good candidate for realizing the in-memory HDC system.

7.2 Associative memory search with CIM

Classification in HDC involves an AM search between the prototype hypervectors and the query hypervector using a suitable similarity metric, such as the inverse Hamming distance (***invHamm***) computed from Equation 3. Using associativity of addition operations, the expression in Equation 3 can be decomposed into the addition of two dot product terms as shown in right side of Equation 3.

$$Class_{Pred} = argmax_{i\in 1,\dots,c} Q \cdot P_i + \overline{Q} \cdot \overline{P_i}$$
$$\approx argmax_{i\in 1,\dots,c} Q \cdot P_i \quad (3)$$

where $\overline{\boldsymbol{Q}}$ denotes the logical complement of $\boldsymbol{Q}$. Since the operations associated with HDC ensure that both the query and prototype hypervectors have an almost equal number of zeros and ones, the dot product (***dotp***) can also serve as a viable similarity metric.

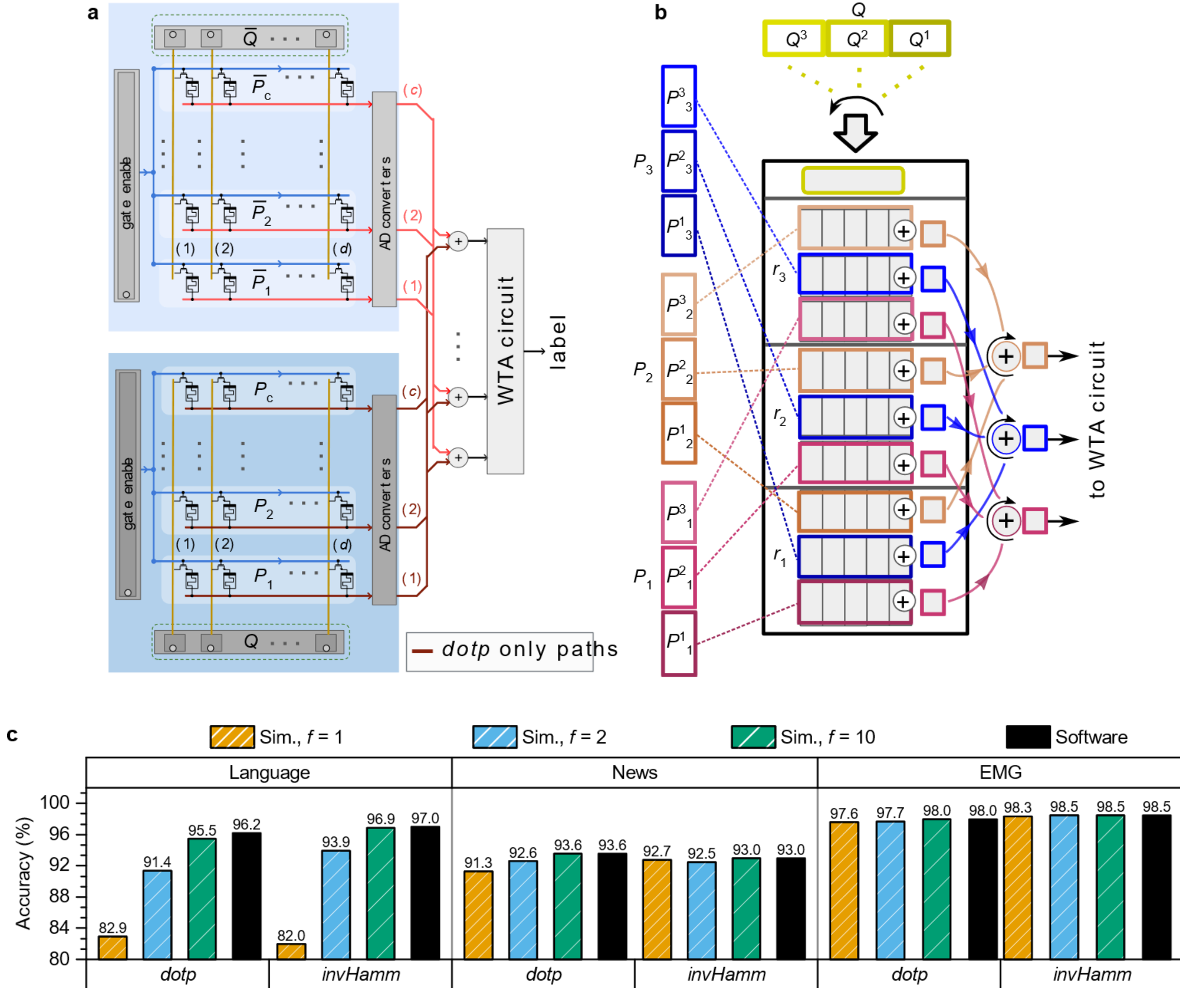


*Figure 19: **Associative memory search** – **a.** Schematic illustration of the AM search architecture to compute the **invHamm** similarity metric. Two PCM crossbar arrays of **c** rows and **d** columns are employed. **b.** Schematic illustration of the coarse-grained randomization strategy employed to counter the variations associated with the crystalline PCM state. **c**. Results of the classification task show that 10-partition simulation accuracy results compare favourably with the software baseline for both similarity metrics on the three datasets.*

To compute the ***invHamm*** similarity metric, two memristive crossbar arrays of ***c*** rows and ***d*** columns are required as shown in Figure 19a. The prototype hypervectors, $\mathbf{P_i}$, are programmed into one of the crossbar arrays as conductance states. Binary `1' components are programmed as crystalline states and binary '0' components are programmed as amorphous states. The complementary hypervectors $\overline{\mathbf{P}_i}$ are programmed in a similar manner into the second crossbar array. The query hypervector $\boldsymbol{Q}$ and its complement $\overline{\boldsymbol{Q}}$ are applied as voltage values along the wordlines of the respective crossbars. In accordance with the Kirchoff's current law, the total current on the ith bitline will be equal to the dot-product between query hypervector and i^{th} prototype hypervector. The results of this in-memory dot-product operations from the two arrays are added in a pairwise manner using a digital adder circuitry in the periphery and are subsequently input to a winner-take-all (WTA) circuit which

outputs a `1' only on the bitline corresponding to the class of maximum similarity value. When ***dotp*** similarity metric is considered, only the crossbar encoding P_i is used and the array of adders in the periphery is eliminated, resulting in reduced hardware complexity.

Simulations and subsequent experiments were performed using a prototype PCM chip to evaluate the effectiveness of the proposed implementation on three common HDC benchmarks: language classification, news classification, and hand gesture recognition from electromyography (EMG) signals. These tasks demand a generic programmable architecture to support different number of inputs, classes, and data types (see the table below).

Dataset	Input type	Size of ***n***	# Channels	Item Memory (IM)		Associative Memory (AM)	
				# Symbols	Dimensionality	Dimensionality	# Classes
Language	Categorical	4	1	27	10000	10000	22
News	Categorical	5	1	27	10000	10000	8
EMG	Numerical	5	4	4	10000	10000	5

While HDC is remarkably robust to random variability and device failures, deterministic spatial variations in the conductance values could pose a challenge. Unfortunately, in our prototype PCM chip, the conductance values associated with the crystalline state do exhibit a deterministic spatial variation. However, given the holographic nature of the hypervectors, this can be addressed by a random partitioning approach. We employed a coarse grained randomization strategy where the idea is to segment the prototype hypervector and to place the resulting segments spatially distributed across the crossbar array (see Figure 19b). This helps all the components of prototype hypervectors to uniformly mitigate long range variations. The proposed strategy involves dividing the crossbar array into ***f*** equal sized partitions $(R_1, \ldots, R_f)$ and storing a ***1/f*** segment of each of the prototype hypervectors per partition. Here ***f*** is called the *partition factor* and it controls the granularity associated with the randomization. To match the segments of prototype hypervectors, the query vector is also split into equal sized subvectors $(Q_1, \ldots, Q_f)$ which are input sequentially to the wordline drivers of the crossbar.

The programming methodology followed to achieve the coarse grained randomized partitioning in memristive crossbar for AM search is explained in the following steps:

- We split all prototype hypervectors $(P_1, \ldots, P_C)$ into ***f*** subvectors of equal length where ***f*** is the partition factor. For example, subvectors from the prototype hypervector of the first class are denoted as: $(P_1^1, \ldots, P_1^f)$.
- The crossbar array is divided into ***f*** equal sized partitions $(R_1, \ldots, R_f)$. Each partition must contain ***D/f*** rows and ***C*** columns.
- A random permutation ***E*** of numbers 1 to ***C*** is then selected.
- The first subvector from each class $(P_1^1, \ldots, P_C^1)$ is programmed into the first partition R_1 such that each subvector fits to a column in the crossbar partition. The order of programming of subvectors into the columns in the partition is determined by the previously selected random permutation E.
- The above steps must be repeated to program all the remaining partitions $(R_1, \ldots, R_f)$.

The methodology followed in feeding query vectors during inference is detailed in the following steps:

- We split query hypervector ***Q*** into ***f*** subvectors $(Q^1, \ldots, Q^f)$ of equal length.

- For $j \in (1, \ldots, f)$ we translate Q^jcomponent values into voltage levels and apply onto the wordline drivers in the crossbar array. Bitlines corresponding to the partition R_j are enabled.
- Depending on the belonging class, the partial dot products are then collected onto respective destination in sum buffer through the A/D converters at the end of R_j partition of the array. The above procedure is repeated for each partition.
- Class-wise partial dot products are accumulated together from each iteration and updated in the sum buffer.
- After the $\boldsymbol{f^{th}}$ iteration, full dot product values are ready in the sum buffer. The results are then compared against each other using a WTA circuit to find the maximum value to assign its index as the predicted class.

A statistical model that captures the spatio-temporal conductivity variations was used to evaluate the effectiveness of the coarse-grained randomized partitioning method. Simulations were carried out for different partition factors 1, 2 and 10 for the two similarity metrics ***dotp*** and ***invHamm*** as shown in Figure 19c. In statistical model simulations, the prototype hypervectors (and their complements) are learned beforehand in software and are then programmed into the PCM devices on the chip. Inference is then performed with a software encoder and using Equation 3 for the associative memory search, in which conductance values are sampled from the an empirical spatial and temporal distribution. The software encoder was employed to precisely assess the performance and accuracy of the associative memory search alone when implemented in hardware.

The simulation results indicate that the classification accuracy increases with the number of partitions. For instance, for language classification, the accuracy improves from 82.5% to 96% with ***dopt*** by randomizing with a partition factor of 10 instead of 1. The 10-partition simulation accuracy (performed with a partition factor of 10) is close to the software baseline for both similarity metrics on all three datasets. When the two similarity metrics are compared, ***invHamm*** provides slightly better accuracy for the same partition size, at the expense of almost doubled area and energy consumption. Therefore, for low-power applications, a good trade-off is the use of ***dotp*** similarity metric with a partition factor of 10.

7.3 N-gram encoding with CIM

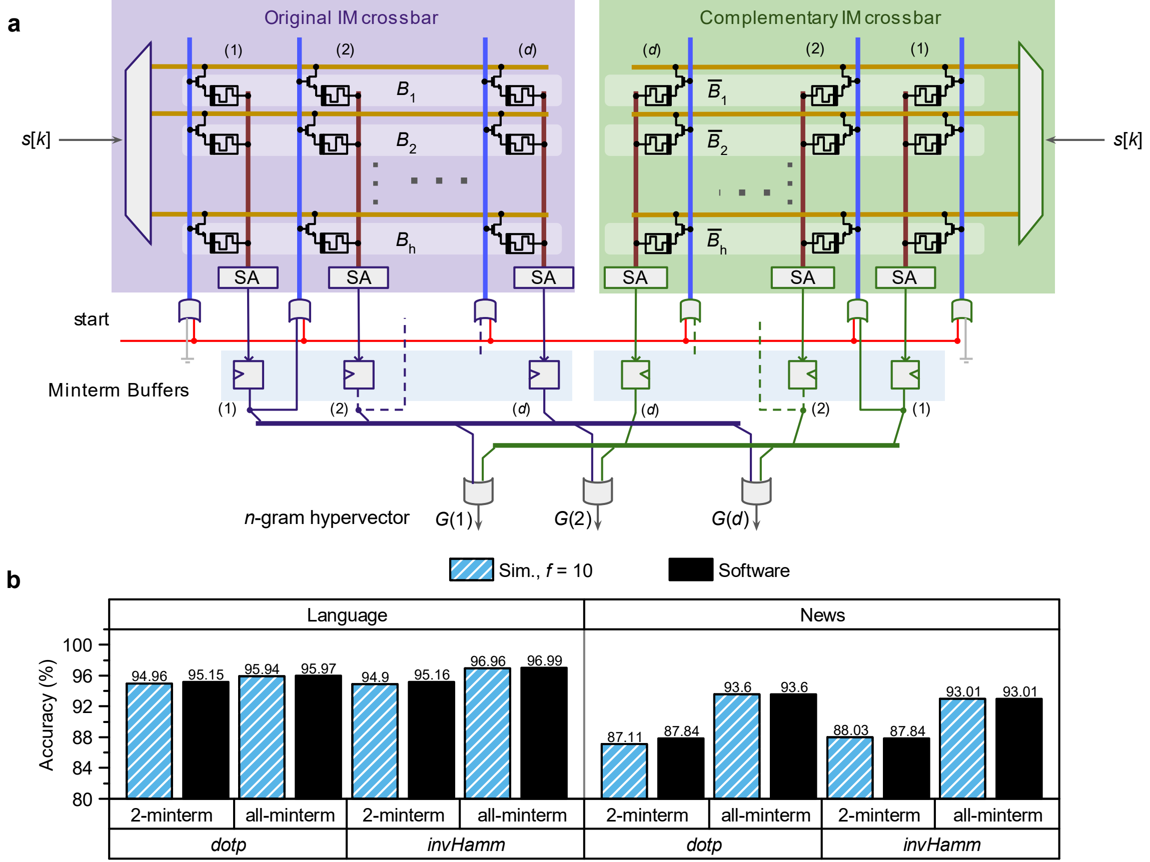


*Figure 20: **In-memory n-gram encoding based on 2-minterm: a.** The basis hypervectors and their complements are mapped onto two crossbar arrays. Through a sequence of in-memory logical operations the approximated **n**-gram **G** as in Equation 5 is generated.**b.** Classification results on the language (using n=4) and news (using n=5) datasets show the performance of the 2-minterm approximation compared with the all-minterm approach.*

In this section, we will focus on the design of the ***n***-gram encoding module. One of the key operations associated with the encoder is the calculation of the ***n***-gram hypervector given by Equation 1. In order to find in-memory hardware friendly operations, Equation 1 is re-written as the component-wise summation of 2^{n-1} minterms given by Equation 4.

$$G = \bigvee_{j=0}^{2^{n-1}-1} L_{1,j}(B[1]) \wedge \rho L_{2,j}(B[2]) \wedge \ldots \wedge \rho^{n-1} L_{n,j}(B[n]) \qquad (4)$$

The operator $L_{k,j}$ is given by: $L_{k,j}(B[k]) = B[k] if (-1)^{Z(k,j)} = 1$
$= \overline{B[k]}\ otherwise$

where $Z(k,j) = \left\lfloor \frac{1}{2^k}(2j + 2^{k-1}) \right\rfloor, k \in \{1,2,\ldots,n\}$ is the item hypervector index within an ***n***-gram and $j \in \{0,1,\ldots,2^{n-1}-1\}$ is used to index minterms. The representation given by Equation 4 can be mapped into memristive crossbar arrays where bitwise AND ($\wedge$) function can be realized using an in-memory read logic operation. However the number of minterms (2^{n-1}) rises exponentially with the size ***n*** of the ***n***-gram, making the hardware computations

costly. Therefore, it is desirable to reduce the number of minterms and to use a fixed number of minterms independent of ***n***.

Based on Equation 4, we empirically obtained a 2-minterm encoding function for calculating the ***n***-gram hypervector given by

$$\hat{G} = (B[1] \wedge \rho B[2] \wedge \ldots \rho^{n-1} B[n]) \vee (\overline{B[1]} \wedge \rho \overline{B[2]} \wedge \ldots \rho^{n-1} \overline{B[n]}) \quad (5)$$

Encoding based on $\hat{G}$ shows significant functional equivalence with the ideal XNOR-based encoding scheme in certain key attributes such as similarity between the basis and prototype hypervectors. A schematic illustration of the corresponding ***n***-gram encoding system is presented in Figure 20. The basis hypervectors are programmed on one of the crossbars and their complement vectors are programmed on the second. The component-wise logical AND operation between two hypervectors in Equation 5 is realized in-memory by applying one of the hypervectors as the gate control lines of the crossbar, while selecting the wordline of the second hypervector. The result of the AND function from the crossbar is passed through an array of sense amplifiers (SA) to convert the analog values to binary values. The binary result is then stored in the minterm buffer, whose output is fed back as the gate controls by a single component shift to the right (left in the complementary crossbar). This operation approximates the permutation operation in Equation 5 as a 1-bit right-shift instead of a circular 1-bit shift. By performing these operations ***n*** times, it is possible to generate the ***n***-gram. After ***n***-gram encoding, the generated ***n***-grams are accumulated and binarized with a threshold that depends on ***n***.

In order to generate a ***n***-gram hypervector in ***n*** cycles, the crossbar is operated using the following procedure (refer to Figure 20):

- During the first cycle, ***n***-gram encoding is initiated by asserting the *start* signal while choosing the index of ***n***th symbol $s[n]$.
- This enables all the gate lines in both crossbar arrays and the wordline corresponding to $s[n]$ to be activated.
- The current released onto the bitlines passed through the sense amplifiers should ideally match the logic levels of $B[n]$ in first array and $\overline{B[n]}$ in the second array.
- The two *'minterm buffers'* downstream of the sense amplifier arrays register the two hypervectors by the end of the first cycle. During subsequent jth ($1 < j \leq n$) cycles, the gate lines are driven by the right shifted version of the incumbent values on the minterm buffers---effectively implementing permutation---while the row decoders are fed with symbol $s[n-j+1]$; the left shift is used for the second crossbar.
- This ensures that the output currents on the bitlines correspond to the component-wise logical AND between the permuted minterm buffer values and the next basis hypervector $B[n-j]$ (complement for the second array).
- The expression for the value stored on the left-side minterm buffers at the end of ***j**th* cycle is given by $\prod_{k=1}^{j} \rho^{j-k} B[n-k+1]$. The product of the complementary hypervectors $\prod_{k=1}^{j} \rho^{j-k} \overline{B[n-k+1]}$ is stored in the right-side minterm buffers.

- At the end of the nth cycle, the two minterms are available in the minterm buffers.
- The elements in the minterm buffers are passed onto the OR gate array following the minterm buffers, such that inputs to the array have matching indices from the two minterm vectors. At this point, the output of the OR gate array reflects the desired ***n***-gram hypervector from 2-minterm ***n***-gram encoding.
- After ***n***-gram encoding, the generated ***n***-grams are accumulated and binarized. The threshold applied to binarize the sum hypervector components is given by:

  $$l \cdot (\frac{1}{2^{n-log(k)}})$$

  where ***l*** is the length of the sequence, ***n*** is the ***n***-gram size, and ***k*** is the number of minterms used for the binding operation in the encoder.

To test the effectiveness of the encoding scheme with in-memory computing, simulations were carried out using the PCM statistical model. The training was performed in software with the same encoding technique used thereafter for inference, and both the encoder and AM were implemented with modelled PCM crossbars for inference. The simulations were performed only on the language and news classification datasets, because for the EMG dataset the hypervectors used for the ***n***-gram encoding are generated by a spatial encoding process and cannot be mapped entirely into a fixed IM of reasonable size. From the results presented in Figure 20b, it is clear that the all-minterm approach to encoding provides the best classification accuracy in most configurations of AM as expected. However, the 2-minterm based encoding method yields a stable and, in some cases, particularly in language dataset, similar accuracy level to that of the all-minterm approach, while significantly reducing the hardware complexity.

8. Application outlook

8.1 Summary of MNEMOSENE applications

All the applications studied in WP1, the associated CIM kernels they rely on, as well and estimated performance and area benefits of the CIM implementation are summarized in Table 5.

Table 5: Summary of MNEMOSENE applications and their benefits

Application	CIM kernel	Performance benefit	Area benefit
Database query	Bulk bitwise AND/OR with scouting logic	Up to 15x faster and 5x-60x lower energy than Intel Xeon multi-core	NA
Matching with automata processor	STE matrix (n-bit input/ 1-bit output)	6x higher throughput than HARE ASIC design	NA
Guided image filtering	Address calculation	100x less address transfer	NA
Compressed sensing	Analog matrix-vector multiplication (8-bit input/output)	80x lower power than 4-bit FPGA at same speed	NA
Deep learning inference	Analog matrix-vector multiplication (4-bit input/output)	1000x lower energy than near threshold Cortex-M processor	37x area saving against 8-bit CIM accelerator
Hyperdimensional computing	Bitwise AND, binary matrix-vector multiplication (1-bit input, 8-bit output)	6x lower energy than equivalent 65-nm CMOS ASIC	3.7x lower area than equivalent 65-nm CMOS ASIC

In general, the CIM implementations are shown to achieve significant benefits in energy and area with respect to alternate implementations that do not use CIM. 5-10x energy benefits are estimated compared with conventional digital CMOS ASICs, and 10-1000x energy benefits are estimated compared with traditional general-purpose processors. In terms of area, the CIM approach also shows some promising advantages compared with CMOS ASICs. Better improvements are generally obtained from the applications using the analog matrix-vector multiplication kernel, which is also more challenging in terms of circuit implementation and memristive device design.

8.2 Other applications that could benefit from MNEMOSENE kernels

The computation kernels developed by the MNEMOSENE project can be applied in different application segments which have extreme demand in terms of storage, energy and computation efficiency. CIM kernels can be used to perform arithmetic (e.g., vector-vector multiplication) operations. This subsection presents some of the application domains in which MNEMOSENE CIM kernels can be applied.

8.2.1 Sparse coding

Sparse coding of information is a powerful means to perform feature extraction on high dimensional data and it is of vital importance for wide range of application segments such as object recognition, computer vision, signal processing and etc. Sparse coding enables to process large amount of data with minimal resources. Thus, sparse coding enables biological neurons to effectively process complex data while consuming very little power. Similarly, sparse coding can be used to implement energy-efficient bio-inspired neuromorphic applications. Since sparse coding mainly rely on bulky matrix-vector multiplication operation, it can directly benefit from the kernels developed in MNEMOSENE project to accelerate the matrix-vector multiplication operation in an efficient manner.

8.2.2 Threshold Logic

Threshold logic is a basic operation that uses a threshold gate which takes *n* inputs (x_1,x_2, …x_n) and generates single output y. A threshold logic has a threshold θ and each input x_i is associated with a weight w_i . A threshold logic unit computes the function output y as follows:

$$y = \begin{cases} 1, & if \sum_{i=1}^{n} w_i x_i \geq \theta \\ 0, & otherwise \end{cases}$$

Equation 6: Threshold logic function

Thus threshold logic determines the output by comparing the weighted sum of its inputs with the defined threshold value. In a neurobiology and neuromorphic context threshold logic mimics the electrical (firing) mechanism of neurons as a result it has been adopted widely in various artificial intelligence applications.

From Equation 6 we can observe that the core operation involved in threshold logic is weighted sum operation or vector-vector multiplication in other words. Therefore, since the vector-vector multiplication operation can be easily mapped to memristive based crossbar array, a threshold logic and threshold logic-based applications can be directly benefit from the kernels developed under MNEMOSENE project.

8.2.3 Linear equation solvers

The vector-matrix multiplication kernel investigated within Mnemosene is a data intensive and highly parallelizable computation and serves as the core operation of various applications such as machine learning, image and signal processing applications. In scientific computing applications, it can also be used in order to solve systems of linear equations. Iterative linear solvers such as Conjugate Gradient perform multiple matrix-vector multiplications on the same matrix A in order to solve Ax = b for x. These multiplications can therefore be performed with CIM, and an iterative refinement algorithm can be implemented in high precision in order to obtain an arbitrarily accurate solution, despite the low precision of the CIM computations [37].

9. Bibliography

[1] F. Xiong and al., *Science,* vol. 332, p. 568, 2011.

[2] K.-S. Li and al., in *Proc. of the Symposium on VLSI Technology*, 2014.

[3] J. J. Yang and al., *Nature Nanotechnology,* vol. 8, p. 13, 2013.

[4] I. Vourkas and al., *IEEE Circuits and Systems Magazine,* vol. 16, p. 15, 2016.

[5] S. Li and al., in *Proc. of the Design Automation Conference (DAC)*, 2016.

[6] L. Xie and al., in *Proc. of the Computer Society Annual Symposium on VLSI (ISVLSI)*, 2017.

[7] S. Hamdioui and al., in *Proc. of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.

[8] Y. Liu and al., in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2016.

[9] H. Li and al., in *IEEE International Electron Devices Meeting (IEDM)*, 2016.

[10] W.-H. Chen and al., in *Proc. of the International Electron Devices Meeting (IEDM)*, 2017.

[11] D. Dua and C. Gra, "UCI machine learning repository," 2017.

[12] N. Chandoke and al., in *Proc. of the International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, 2015.

[13] V. G. M. L. B. I. H. S. P. C. D. M. E. E. Joshi, "Accurate deep neural network inference using computational phase-change memory," *http://arxiv.org/abs/1906.03138,* 1-25 March 2019.

[14] F. C. J. M. L. S. H. L. Y. B. V. Z. Z. L. W. D. Cai, "A fully integrated reprogrammable memristor–CMOS system for efficient multiply–accumulate operations," *Nature Electronics, 2(7),* pp. 290-299, 2019.

[15] M. W. R. S. S. J. P. L. Z. G. E. M. D. N. Y. J. J. Hu, "Dot-product engine for neuromorphic computing," *Proceedings of the 53rd Annual Design Automation Conference on - DAC '16,* 2016.

[16] O. G. R. H.-T. J. A. D. M. P. H. R. I. V. C. Banos, "mHealthDroid: A Novel Framework for Agile Development of Mobile Health Applications," *https://doi.org/10.1007/978-3-319-13105-4_14,* 2014.

[17] Z.-G. &. M. M. Liu, "Learning low-precision neural networks without Straight-Through Estimator(STE)," *http://arxiv.org/abs/1903.01061,* 2019.

[18] S. D. N. H. A. T. M. S. A. G. M. L. P. S. B. L. Hamdioui, "Applications of Computation-In-Memory Architectures based on Memristive Devices," *Design, Automation & Test in Europe Conference & Exhibition (DATE),* 2019.

[19] P. Kanerva, "Binary Spatter-Coding of ordered k-tuples," in *Proceedings of the International Conference on Artificial Neural Networks (ICANN)*, 1996.

[20] A. Joshi, J. T. Halseth and P. Kanerva, "Language geometry using random indexing," in *International Symposium on Quantum Interaction*, 2016.

[21] L. Chua, "Resistance switching memories are memristors," *Applied Physics A,* vol. 102, p. 765–783, 2011.

[22] H.-S. P. Wong and S. Salahuddin, "Memory leads the way to better computing," *Nature nanotechnology,* vol. 10, p. 191, 2015.

[23] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart and R. S. Williams, "'Memristive' switches enable 'stateful' logic operations via material implication," *Nature,* vol. 464, p. 873, 2010.

[24] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny and U. C. Weiser, "MAGIC–Memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs,* vol. 61, p. 895–899, 2014.

[25] W. Shen, P. Huang, M. Fan, R. Han, Z. Zhou, B. Gao, H. Wu, H. Qian, L. Liu, X. Liu and others, "Stateful Logic Operations in One-Transistor-One-Resistor Resistive Random Access Memory Array," *Electron Device Letters,* vol. 40, p. 1538–1541, 2019.

[26] H. Li, T. F. Wu, A. Rahimi, K. S. Li, M. Rusch, C. H. Lin, J. L. Hsu, M. M. Sabry, S. B. Eryilmaz, J. Sohn, W. C. Chiu, M. C. Chen, T. T. Wu, J. M. Shieh, W. K. Yeh, J. M. Rabaey, S. Mitra and H. S. P. Wong, "Hyperdimensional computing with 3D VRRAM in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition," in *Proceedings of the International Electron Devices Meeting (IEDM)*, 2016.

[27] H. Li, T. F. Wu, S. Mitra and H. S. P. Wong, "Device-architecture co-design for hyperdimensional computing with 3D vertical resistive switching random access memory (3D VRRAM)," in *Proceedings of the International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*, 2017.

[28] T. F. Wu, H. Li, P. Huang, A. Rahimi, J. M. Rabaey, H. .. P. Wong, M. M. Shulaker and S. Mitra, "Brain-inspired computing exploiting carbon nanotube FETs and resistive RAM: Hyperdimensional computing case study," in *Proceedings of the International Solid State Circuits Conference (ISSCC)*, 2018.

[29] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE,* vol. 98, p. 2201–2227, 2010.

[30] G. W. Burr, M. J. Brightsky, A. Sebastian, H.-Y. Cheng, J.-Y. Wu, S. Kim, N. E. Sosa, N. Papandreou, H.-L. Lung, H. Pozidis and others, "Recent progress in phase-change

memory technology," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems,* vol. 6, p. 146–162, 2016.

[31] D. Kuzum, R. G. D. Jeyasingh, B. Lee and H.-S. P. Wong, "Nanoelectronic programmable synapses based on phase change materials for brain-inspired computing," *Nano letters,* vol. 12, p. 2179–2186, 2011.

[32] T. Tuma, A. Pantazi, M. Le Gallo, A. Sebastian and E. Eleftheriou, "Stochastic phase-change neurons," *Nature Nanotechnology,* vol. 11, p. 693, 2016.

[33] I. Boybat, M. Le Gallo, S. R. Nandakumar, T. Moraitis, T. Parnell, T. Tuma, B. Rajendran, Y. Leblebici, A. Sebastian and E. Eleftheriou, "Neuromorphic computing with multi-memristive synapses," *Nature communications,* vol. 9, p. 2514, 2018.

[34] A. Sebastian, M. Le Gallo, G. W. Burr, S. Kim, M. BrightSky and E. Eleftheriou, "Tutorial: Brain-inspired computing using phase-change memory devices," *Journal of Applied Physics,* vol. 124, p. 111101, 2018.

[35] P. Hosseini, A. Sebastian, N. Papandreou, C. D. Wright and H. Bhaskaran, "Accumulation-based computing using phase-change memories with FET access devices," *Electron Device Letters,* vol. 36, p. 975–977, 2015.

[36] A. Sebastian, T. Tuma, N. Papandreou, M. Le Gallo, L. Kull, T. Parnell and E. Eleftheriou, "Temporal correlation detection using computational phase-change memory," *Nature Communications,* vol. 8, p. 1115, 2017.

[37] M. Le Gallo, A. Sebastian, R. Mathis, M. Manica, H. Giefers, T. Tuma, C. Bekas, A. Curioni and E. Eleftheriou, "Mixed-precision in-memory computing," *Nature Electronics,* vol. 1, p. 246, 2018.

[38] D. Ielmini and H.-S. P. Wong, "In-memory computing with resistive switching devices," *Nature Electronics,* vol. 1, p. 333, 2018.